NPS ARCHIVE
2000.03
KATI, E.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

FAULT-TOLERANT APPROACH FOR DEPLOYING SERVER AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM) SERVER IN WINDOWS NT ENVIRONMENT TO PROVIDE UNINTERRUPTED SERVICES TO ROUTERS INCASE OF SERVER FAILURE(S)

by

Efraim KATI

March 2000

Thesis Advisor: Geoffrey Xie
Second Reader: James Bret Michael

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE: FAULT TOLERANT APPROACH FOR DEPLOYING SERVER AGENT BASED ACTIVE NETWORK MANAGEMENT (SAAM) SERVER IN WINDOWS NT ENVIRONMENT TO PROVIDE UNINTERRUPTED SERVICES TO ROUTERS INCASE OF SERVER FAILURE(S).

5. FUNDING NUMBERS

6. AUTHOR(S) Kati, Efraim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA and NASA

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
G417

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
Statement A

13. ABSTRACT *(maximum 200 words)*

With the explosive growth of the Internet and high demand on real-time network applications, the need for integrated service networks has emerged. Therefore, the Next Generation Internet project and as a part of this project the Server Agent-based Active network Management (SAAM) project was initiated. SAAM is a server based hierarchical routing architecture designed to provide Quality of Service routing services for network resource intensive applications. In the SAAM architecture, a small number of dedicated SAAM servers perform most of the network management tasks on behalf of the routers. The SAAM server has a great responsibility in the SAAM architecture and failure of the SAAM server can have a devastating effect on the performance of the entire network. In order to tolerate the failure of the SAAM server, this thesis examines the fault tolerance (ft.) for the SAAM server in two phases: local area ft., and remote area ft. For the local area ft., after a survey of the literature and commercial offerings, a recommended solution is proposed. For the remote area ft., a backup server model is designed and prototyped. The prototyped model provides robust error detection and a fast recovery from the failure of the primary SAAM server.

14. SUBJECT TERMS Fault Tolerance, Heartbeat Protocol, Next Generation Internet, Networks	15. NUMBER OF PAGES 328
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

FAULT-TOLERANT APPROACH FOR DEPLOYING SERVER AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM) SERVER IN WINDOWS NT ENVIRONMENT TO PROVIDE UNINTERRUPTED SERVICES TO ROUTERS INCASE OF SERVER FAILURE(S)

Efraim KATI
First Lieutenant, Turkish Army
B.S., Turkish Military Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

NPS ARCHIVE
2000 03
KAT I, E.

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The current data networks are mainly based on sophisticated stand-alone routers that provide best effort service. However, with the explosive growth of the Internet and high demand on real-time network applications, the need for integrated service networks has emerged. For this purpose the Next Generation Internet (NGI) Project and as a part of this project the Server Agent based Active network Management (SAAM) project was initiated. SAAM is a server based hierarchical routing architecture designed to provide Quality of Service (QoS) routing services for network resource intensive applications. In the SAAM architecture, a small number of dedicated SAAM servers perform most of the network management tasks on behalf of the routers. The SAAM server has a great responsibility in the SAAM architecture and failure of the SAAM server can have a devastating effect on the performance of the entire network. In order to tolerate the failure of the SAAM server and provide uninterrupted services to routers, this thesis examines the fault tolerance for the SAAM server in two phases: local area fault tolerance, and remote area (disaster recovery) fault tolerance. For the local area fault tolerance, after a survey of the literature and commercial offerings, a recommended solution is proposed. For the remote area fault tolerance, a backup server model is designed and prototyped. The prototyped model provides robust error detection and a fast recovery from the failure of the primary SAAM server.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. OVERVIEW OF SAAM 2

C. PURPOSE OF THIS THESIS 5

D. SCOPE OF THIS THESIS 6

E. ORGANIZATION OF THIS THESIS 6

II. OVERVIEW OF FAULT TOLERANCE 9

A. BASIC CONCEPTS AND DEFINITIONS 9

B. REDUNDANCY CONCEPT 12

1. Hardware Redundancy 13

a. Passive Hardware Redundancy 13

b. Active Hardware Redundancy 15

2. Software Redundancy 18

a. Consistency Checks 19

b. Capability Checks 19

c. N-Version Programming 19

d. Recovery Blocks 20

3. Information Redundancy 21

4. Time Redundancy 22

C. OBJECTIVES OF FAULT TOLERANCE 22

1. Dependability 23

2. Reliability 23

3. Availability 25

4. Safety 26

5. Performability 27

6. Maintainability 27

7. Testability 28

D. PHASES IN FAULT TOLERANCE 28

1. Error Detection 29

a. Replication Checks 30

b. Timing Checks 31

c. Structural Checks 31

d. Reasonableness Checks 32

e. Diagnostics Checks 32

2. Damage Confinement and Assessment 32

3. Error Correction 33

a. Error Recovery 33

b. Error Masking 34

4. Fault Treatment and Continued Service 34

III. FAULT TOLERANCE IN WINDOWS NT OPERATING SYSTEM....37
A. ERROR HANDLING AND PROTECTED SUBSYSTEMS....38
B. NT FILE SYSTEM (NTFS)....39
C. AUTOMATIC RESTART....40
D. TAPE BACKUP SUPPORT....42
E. UNINTERRUPTIBLE POWER SUPPLY (UPS)....43
F. FAULT-TOLERANT STORAGE....44
1. Stripe Set....46
2. Mirror Set....46
3. Stripe Set With Parity....47
G. MICROSOFT CLUSTER SERVER (MSCS)....49
1. Overview of Server Cluster....49
2. MSCS....50
H. WINDOWS NT LOAD BALANCING SERVICE (WLBS)....56
1. WLBS Features....57
2. WLBS Shortcomings....59
IV. LOCAL AREA FAULT TOLERANCE FOR SAAM SERVER....61
A. PRODUCTS OVERVIEW....61
1. ARCserve Replication 4.0 for Windows NT....62
2. Co-Standby*Server* 4.2 for Widows NT....66
3. Double-Take 3.0....71
4. Endurance 4000....82
5. Octopus 3.2....89
B. RECOMMENDATIONS....93
V. REMOTE AREA FAULT TOLERANCE FOR SAAM SERVER....97
A. MODELING....97
1. Server States....98
2. Failure Detection....100
a. Constant Heartbeat Protocol (V.0)....102
b. Accelerated Heartbeat Protocol (V.1)....104
c. Prototyping of the Heartbeat Protocols....107
d. Performance Comparison of the Heartbeat Protocols....119
e. Preventing False Failure Detection....128
f. Preventing Late Failure Detection....133
g. Existence of Two Active Servers at the Same Time....135
3. Damage Confinement and Assessment....136
4. Failure Recovery....139
a. Size of PIB Data Records....140
b. Selected Approach....144

5. Fault Treatment and Continued Service 145
B. INTEGRATION WITH THE EXISTING SOURCE SODE 147
1. Packet Formats 147
2. Integration of Error Detection Mechanism 150
a. `HeartbeatQuery` Class 151
b. `HeartbeatResponse` Class 152
c. `HeartbeatController` Class 153
d. `BannerFrame` Class 154
3. Modifications Done on the Existing Source Code 155
a. `Message` Class 155
b. `Server` Class 156
c. `ServerAgent` Class 158
d. `PacketFactory` Class 158
C. TESTING 159
1. Testbed 159
2. Tests Performed 161
a. Failure Detection Test 162
b. Heartbeat Response Message Loss Test 165
c. Message Numbering Scheme Test 168
d. Unsolicited Heartbeat Test 173
e. Control Channel Auto-configuration Test 175
VI. CONCLUSIONS 179
A. SYNOPSIS AND CONCLUSION 179
B. FUTURE WORK 180
1. Testing of Recommended COTS-Based Product 181
2. Detection of Backup SAAM Server Failures 181
3. Reinstating of a Repaired Server 182
4. Handling of Two Simultaneously Active Servers 182
5. Field Test 182
6. Alert Mechanism 183
APPENDIX A. THE CONSTANT HEARTBEAT PROTOCOL SOURCE FILES 185
APPENDIX B. THE ACCELERATED HEARTBEAT PROTOCOL SOURCE FILES 207
APPENDIX C. NEWLY ADDED SOURCE FILES FOR INTEGRATION 241
APPENDIX D. MODIFIED SOURCE FILES FOR INTEGRATION 259
LIST OF REFERENCES 307
INITIAL DISTRIBUTION LIST 311

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENT/DEDICATIONS

The author would like to acknowledge the Defense Advance Research Projects Agency (DARPA) and the National Aeronautics and Space Agency (NASA) for sponsoring this thesis research.

I would like to extend my sincere gratitude to my thesis advisor, Professor Geoffrey Xie and Professor James Bret Michael, for their patience. I would like to thank my wife, Ozlem Kati for her love and support during the entire thesis process. I would also like to thank to my parents, Arife and Sakir Kati, without whose love and sacrifice I would not have the opportunities that I am enjoying today. And finally, I would like to dedicate this thesis to my son, Efekan Ozgur Kati for his unconditional love.

I. INTRODUCTION

A. BACKGROUND

With the explosive growth of the Internet and increasing demand on real-time network applications, the need for an *integrated services network* has emerged. An integrated services network supports all types of data using a single network and must meet two requirements. First, an integrated services network must guarantee Quality of Service (QoS) to individual user sessions. Second, an integrated services network must support real-time applications that have very stringent packet delay requirements. [Ref. 35] However, the current data networks are mainly based on sophisticated stand-alone routers that provide best-effort service. Therefore, current data networks are not capable of supporting integrated services. To solve this problem, the Next Generation Internet (NGI) initiative, an advanced research program, was initiated. Specifically, the NGI initiative fosters partnerships among academia, industry, and Federal laboratories to develop and experiment with technologies that will enable more powerful and versatile information networks of the 21st century.

One proposal developed under the NGI initiative is the SAAM project. [Ref. 35] SAAM stands for Server Agent-based Active network Management. The SAAM project is currently sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Aeronautics and Space Administration (NASA), and is an ongoing project. More information on the SAAM project can be obtained from the project's World Wide Web home page (www.saamnet.org).

B. OVERVIEW OF SAAM

SAAM is a network management system that enables a network to provide integrated services. Instead of a totally router-based architecture, SAAM utilizes a server-based hierarchical routing architecture that provides Quality of Service (QoS) routing services for network resource intensive applications.

Compared to the current shortest path algorithms, QoS-based routing algorithms must deal with more constraints. Therefore, QoS-based routing requires more processing power at each router. Due to this processing power requirement, when the QoS-based routing is implemented, current sophisticated stand-alone routers can easily become performance bottlenecks. However, SAAM relieves individual routers from most routing and management tasks by employing a small number of dedicated SAAM servers to perform these tasks on behalf of the routers. Such a lightweight router approach implemented by SAAM increases the performance of the routers to support QoS-based routing for real-time applications.

The SAAM server maintains an accurate picture of the QoS capabilities of the network by periodically retrieving link performance information from the routers, and aggregating this information into a ready-to-use database of useful paths. This database is called the *Path Information Base* (PIB) (shown in Figure 1.1). By using the PIB, the SAAM server can efficiently implement network functions such as QoS routing and re-routing of real-time flows, which are required for providing integrated services.

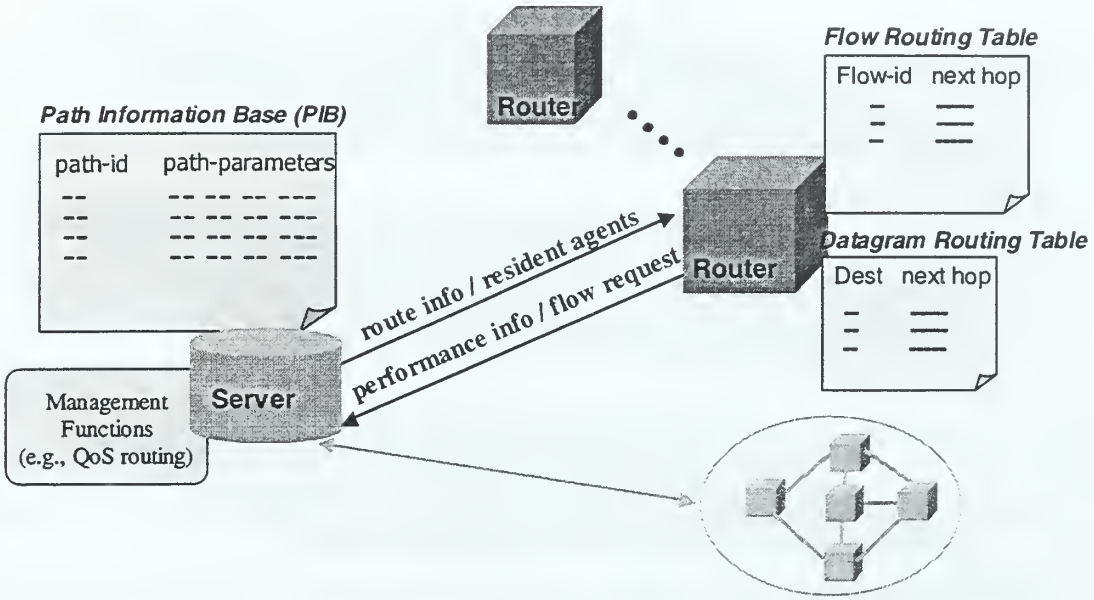


Figure 1.1. Logical Model of SAAM.

To make its service scalable for large networks, SAAM organizes its SAAM servers in a hierarchy, as shown in Figure 1.2. At the first level of the hierarchy, SAAM partitions the network into autonomous regions. These regions are called SAAM regions. SAAM assigns one SAAM server for each SAAM region. At the first level, each SAAM server is responsible for collecting link performance information from routers in its own region and summarizing this collected information for a higher-level server. In the second level of the hierarchy, the child servers periodically send the summarized path performance information to the parent server. In each SAAM region, there is subset of routers, called border gateways, responsible for traffic in and out of this region. The parent server determines the routing cross-regions.

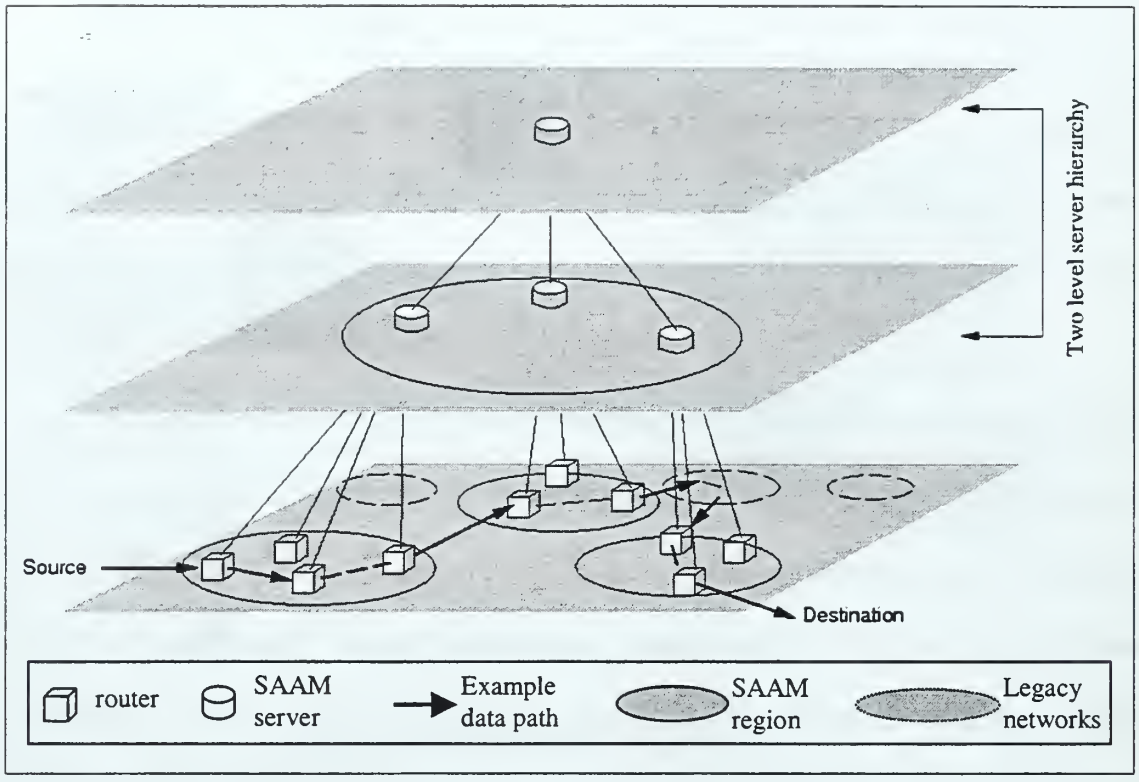


Figure 1.2. Hierarchical Organization of SAAM Servers. [From Ref. 36]

The Internet currently consists of many independently operated Internet Service Providers (ISPs). In Figure 1.2, the regions are considered as ISPs. SAAM is completely compatible with the legacy networks and supports existing inter-domain routing protocols. The border gateway routers make the necessary translation between the different domain protocols. Therefore, whether or not an ISP is using SAAM is transparent to other ISPs. A non-SAAM ISP can still send traffic through a SAAM ISP. Therefore, SAAM can be deployed incrementally, providing improvements of network performance to ISPs that use SAAM. The ISP that uses a SAAM server has total control over the operation of its internal SAAM server. In this case, the parent server only provides performance-enhancing advice to the internal servers. The internal server will verify the advice based on local policies before updating the states of its routers.

C. PURPOSE OF THIS THESIS

A SAAM server is responsible for performing most of the routing and network management tasks on behalf of the routers in its region. Therefore, the quality of the integrated services provided by the network region depends entirely upon the performance of the SAAM server. If not carefully designed, a failure of the SAAM server can have a devastating effect on the performance of the entire network.

The main purpose of this thesis is to add fault tolerance features to the SAAM architecture in order to tolerate server failures. Consequently, a failure of the SAAM server will not interrupt SAAM services to the routers. Fault tolerance features to be added should address the following fault tolerance related requirements of the SAAM server:

- No singe point of failure should be allowed.
- Detection, isolation, and recovery of the failures should happen in seconds (preferably under two seconds).
- Environmental faults such as flood, fire, and earthquake should be addressed.
- Failed SAAM server should be repaired while the system is in operation.
- Reinstating the repaired SAAM server should not affect the provided SAAM services to routers.

D. SCOPE OF THIS THESIS

In order to provide a fault tolerance solution for the SAAM server that best meets the aforementioned requirements, fault tolerance for the SAAM servers is implemented in two phases: local and remote. The first phase, local area fault tolerance for the SAAM server, focuses mainly on tolerating component failures of one server such as processor failure, disk failure and network interface card failure. The second phase, remote area fault tolerance (disaster recovery) for the SAAM server, backup servers are used to tolerate environmental faults such as fire, earthquake, and flood that cause unrecoverable server failures. The function of the second phase is to tolerate the failure of the local area fault tolerance implementation of the SAAM server.

A Commercial-Off-The-Shelf (COTS) based solution is proposed for local area fault tolerance after a survey of the literature and commercial offerings. For remote area fault tolerance, a backup server model is designed and implemented. Additionally, the implemented model is tested in a live SAAM testbed.

E. ORGANIZATION OF THIS THESIS

The remainder of this thesis is organized into the following chapters:

- Chapter II: Overview of Fault Tolerance. Provides an overview of fault tolerance. Also explains basic terminology and the principles of fault tolerance.
- Chapter III: Fault Tolerance in Windows NT Operating System. Explores the fault tolerance related features of the Windows NT operating system.

- Chapter IV: Local Area Fault Tolerance for SAAM Server. Focuses mainly on tolerating component failures of one server. Also examines the five most promising third-party products for the Windows NT operating system and proposes one of these products as a recommended solution.
- Chapter V: Remote Area Fault Tolerance for SAAM Server. Focuses on tolerating environmental faults such as flood, fire, and earthquake. A backup server model is designed and prototyped. Also explains the integration of the prototype with the existing SAAM server source code.
- Chapter VI: Conclusions. Summarizes the test results of the implemented backup server model and outlines the work that remains to be done in the SAAM project.

THIS PAGE INTENTIONALLY LEFT BLANK

II. OVERVIEW OF FAULT TOLERANCE

A. BASIC CONCEPTS AND DEFINITIONS

First the definitions of system, error, fault, and failure will be presented. These terms are used in a variety of ways in different contexts. Although the terms, *fault, failure,* and *error* are generally used interchangeably, they have distinct meanings in fault tolerance terminology. The definitions presented in this section, will clarify the distinctions among the meanings of these terms. Throughout this thesis, the terms, *fault, failure,* and *error* will be used in accordance with the following provided definitions.

The concept of a system is quite general and it is used in other disciplines. In general, a *system* is defined as an identifiable mechanism that maintains a pattern of behavior at an interface between the system and its environment [Ref. 1]. In computer systems, the term *interface* represents identifiable hardware or physical entities. Although a system is considered as a single module, in fact systems are composed of a number of subsystems. Therefore, terminology used for the system under consideration also applies recursively to its subsystems.

Each system has an ideal *specified behavior* and an observed *actual behavior.* A *failure* is a deviation of the actual behavior from the specified behavior [Ref. 2]. An *error* is that part of the system state which is liable to lead to a failure [Ref. 3]. A *fault* is a physical defect, imperfection, or flaw that occurs within some hardware or software component [Ref. 4]. Although a fault has the potential of generating errors, it may not

cause any errors during the period of observation. On the other hand, the existence of an error always indicates that the system has a faulty part.

Faults can be classified using two different key attributes, *duration* and *cause*. The duration specifies the length of the time that a fault is active. When duration is considered, faults are classified as transient, intermittent or permanent. *Transient faults* appear and disappear within a very short period of time. *Intermittent faults* repeatedly appear, but always for a short duration. *Permanent faults* remain in existence indefinitely if no corrective action is taken.

According to their causes, faults are classified as design or operational faults. *Design faults* appear during the system design or modification phases. *Operational faults* appear during the system lifetime and they are caused by physical reasons such as electromagnetic interference, battle damage, operator mistakes, and environmental extremes.

If the system behaves as it is specified, then this state is called *service accomplishment state*. However if the system behaves different from its specifications, then the system enters a state called a *service interruption state*. Usually a system's actual behavior replicates its specified behavior. As shown in Figure 2.1, a fault occasionally creates an error causing a system to fail.

When a system fails, it enters a service interruption state. After the error is detected, reported, and corrected, the system returns to a service accomplishment state. The time between the occurrence of a fault and the appearance of an error is called *fault latency*. The time between the occurrence of an error and the appearance of the resulting

failure is called *error latency*. The total time between the occurrence of a physical failure and the appearance of a failure is the sum of the fault latency and the error latency.

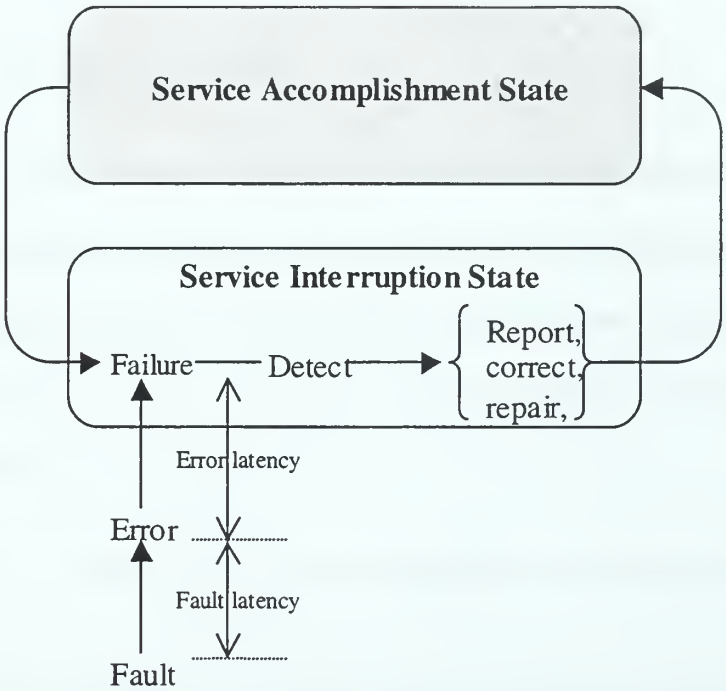


Figure 2.1. Service States of a System. [After Ref. 2]

In order to improve or maintain the system's normal performance (i.e., to keep the system in the service accomplishment state) three techniques are used: fault avoidance, fault masking, and fault tolerance. The *fault avoidance* technique is used for preventing the occurrence of faults. Fault avoidance can include some quality control measures implemented before the system becomes operational, such as design reviews, components screening, and testing. The *fault masking* technique is used for preventing faults in a system from introducing errors. The *fault tolerance* technique is used for preventing system failures from occurring, even if errors caused by faults appear in the system. Since

failures are directly caused by errors, the terms *fault tolerance* and *error tolerance* are often used interchangeably.

A system is considered to be *fault-tolerant* if the actual behavior of the system stays consistent with its specifications, despite the failures of its sub-systems. It is not possible to make a system fault tolerant against its own failures. If the system fails due to an error, then there is nothing that can be done in terms of fault tolerance. However, a system can be designed to be fault tolerant against the failure of its sub-systems. Consequently, the ultimate goal of fault tolerance is to prevent a system failure when some of its sub-systems fail.

B. REDUNDANCY CONCEPT

Redundancy is defined as those parts of the system that are not needed for the correct functioning of the system if no fault tolerance is to be supported [Ref. 1]. Redundancy is the guiding principle behind fault tolerance. In order to build a fault-tolerant system, some redundant sub-systems must exist in the system to be used instead of a failed sub-system. Therefore, redundancy is essential for fault tolerance. On the other hand, redundancy can introduce some side effects into the system. These side effects can take the form of performance degradation, an increase in the size and weight of the system, or reduced reliability*.

* Probability of hardware failure, not the entire system.

Four types of redundancy could be implemented. They are *hardware redundancy*, *software redundancy*, *time redundancy* and *information redundancy*. The following sections will discuss the four redundancy techniques in detail.

1. Hardware Redundancy

Hardware redundancy refers to the replication of the hardware components of the system. As the hardware sizes have become smaller and less expensive, the concept of hardware redundancy becomes more practical. Hardware redundancy can be implemented using one of two techniques, *passive hardware redundancy* and *active hardware redundancy*.

a. Passive Hardware Redundancy

Passive hardware redundancy can be used to mask the occurrence of faults and to prevent the faults from causing the system to fail. This approach does not require any error detection or system reconfiguration. The passive hardware redundancy technique implementations rely upon the voting mechanism among the replicated hardware components, and inherently tolerate the faults.

A simple passive hardware redundancy design can be implemented using three replicated hardware units and a voter, as shown in Figure 2.2. This type of passive redundancy is called *Triple Modular Redundancy* (TMR). In triple modular redundancy, outputs of the three modules are voted and the majority of the output is allowed to pass through the voter. If one of the three modules becomes faulty, the remaining two modules can mask the faulty module.

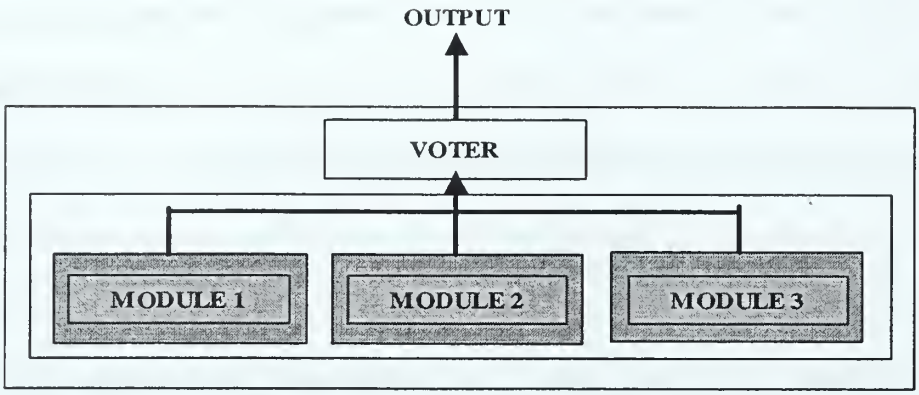


Figure 2.2. Basic Triple Modular Redundancy Design.

The major problem with the TMR is the fact that the voter is a single-point-of-failure. A failure of the voter results in the failure of the system. Thus, the reliability of the system is directly proportional to the reliability of the voter. One approach to overcome this dilemma is to triplex the voters as well as the modules and to provide three independent outputs. A sample design for the voter triplexing approach in TMR is shown in Figure 2.3.

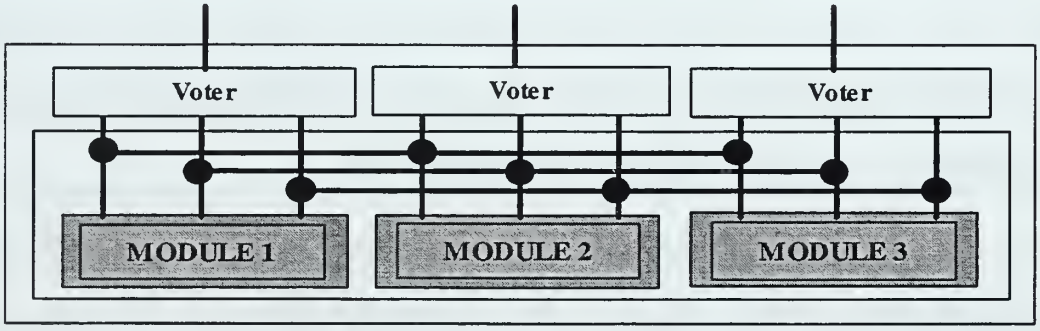


Figure 2.3. Triplexing of the Voters in a TMR Design.

A generalization of the TMR approach is called *N-Modular Redundancy* (NMR). In the NMR technique, N represents the number of replicated modules in the design and it is usually an odd number greater than three so that majority voting can be

used. Although TMR can mask only one faulty component, NMR can mask the effect of more than one faulty modules. For example, in a 5-MR design, it is possible to mask two faulty modules. It can be shown that the number of faulty components that can be masked using the NMR approach equals $(N-1)/2$.

The simple and powerful passive hardware redundancy technique seems to have solved the hardware fault-tolerance problem. It can mask almost all physical device failures. However, it does not mask failures caused by hardware design flaws. If all the modules have faulty designs, then the comparators or voters, no matter how many of them, will not help.

Passive hardware redundancy itself does not improve availability or reliability. In fact, adding redundancy reduces reliability in some designs as explained above. This parallels the airplane analogy: A two-engine airplane costs twice as much and has twice as many engine problems as a one-engine airplane. Redundancy designs require *repair* to dramatically improve availability. [Ref. 2]

b. Active Hardware Redundancy

In active hardware redundancy, fault tolerance is implemented by fault detection, location and recovery instead of by fault masking. In active hardware redundancy, erroneous states are acceptable as long as the system is capable of detecting the errors, reconfiguring itself and regaining its operational state.

The main states of the active redundancy approach are shown in Figure 2.4. Once the fault occurs during normal operation, an error in the system results. If the error is not detected and corrected, the consequence will be system failure. Once the error

is detected, and then the faulty component causing the error has to be located and removed from the system's configuration. Then, the system must be reconfigured with the redundant component instead of the failed one. Finally, the system returns to either its normal operational state or a degraded operational state, depending on the performance of replacement component.

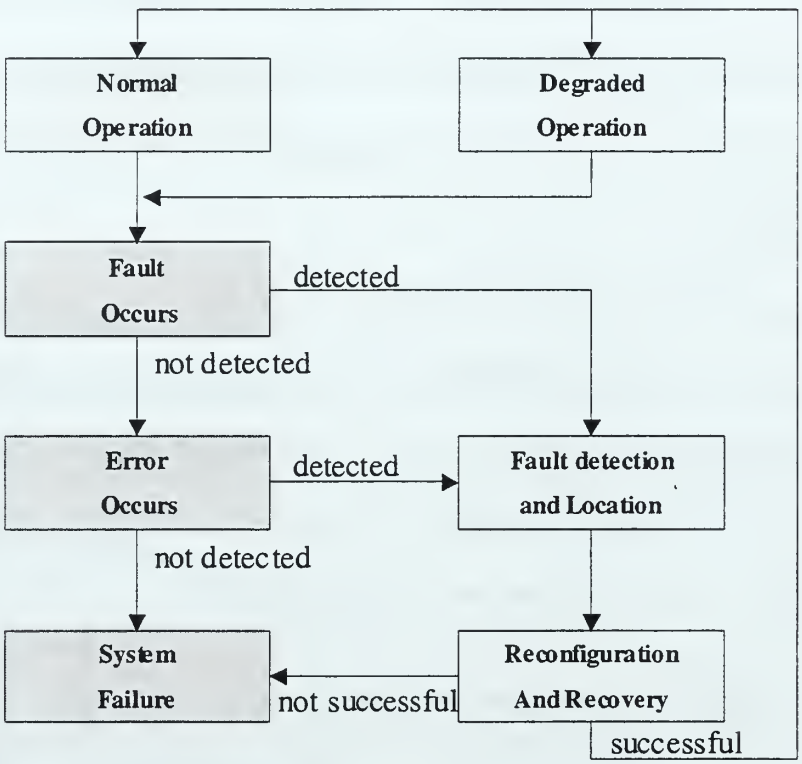


Figure 2.4. Operation of an Active Hardware Redundancy Approach. [After Ref. 4]

In the active hardware redundancy approach, the detection of faults is of great importance. The *duplication with comparison* scheme is an example of a fault detection mechanism that can be used. Duplication with comparison refers to the development of two identical pieces of hardware, having them perform the same computation in parallel, and comparing the results of those computations. If the results of

the two computations do not match, a fault is detected and an error message is generated. Although duplication with comparison does not provide fault tolerance by itself, it is mainly used as a fault detection mechanism in the active hardware redundancy approach.

Once the faulty component is detected and identified, the system should be reconfigured to replace the faulty component. This reconfiguration can be achieved by a technique called *standby replacement* or *standby sparing*. Standby replacement refers to replacing the faulty component with the provided spares.

The standby replacement process introduces a momentary interruption in the service delivered by the system. To minimize this interruption, a form of standby replacement process called *hot standby sparing* can be used. In hot standby sparing, a redundant component operates in parallel with the online component, establishing the readiness for the redundant component to take over in the feature.

There is another form of the standby replacement technique, called *cold standby sparing*. Unlike the hot standby sparing method, the spares in cold standby sparing remain non-operational until they are needed. The process of bringing the spare into an operational state increases the service interruption time. However, power consumption is lower for cold standby sparing than for of hot standby sparing.

A variation of the standby replacement technique is called *pair-and-spare* or *dual-dual*. Basically, the pair-and-spare scheme uses both hot standby sparing and duplication with comparison techniques in its design. It combines two *fail-fast* modules, as shown in Figure 2.5, to produce a super module that continues operating, even if one of the sub modules fails. Fail-fast modules are designed in such a way that they either operate correctly or stop operating immediately. This is achieved by using duplication

with the comparison approach. When the outputs of the modules in the fail-fast module do not match, an error signal is generated and the other module is stopped immediately. Additionally the two fail-fast modules operate in parallel similar to the hot standby sparing method. Because each sub module is fail-fast, the combination is just like the logical "OR" of the two sub modules.

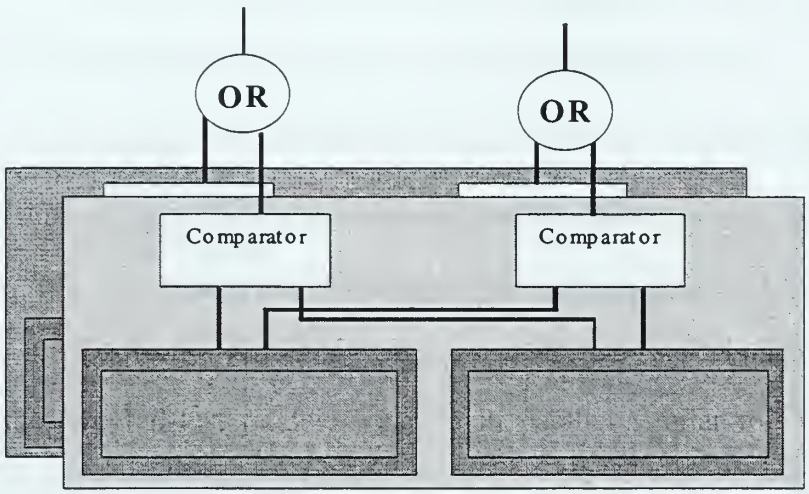


Figure 2.5. Basic Pair-and-Spare Design.

2. Software Redundancy

Redundancy in software can be implemented in many ways—it is not necessary to replicate the complete software program to achieve software redundancy. Software redundancy can appear as several extra lines in the code for checking specific values or as a routine used to periodically test the specific locations in the system's memory. The following sections will discuss some basic software redundancy techniques.

a. Consistency Checks

Consistency checks are used to verify the correctness of specific information in the software application. In some applications, a specific set of data members are required to remain in certain value ranges. If the value of the data exceeds their predetermined value, then an error is declared. Memory address checking mechanisms, implemented in the operating system software, can be given as an example of consistency checks. The memory address checking mechanism determines if the address access generated by the application is outside of the available address range of the memory.

b. Capability Checks

Capability checks are performed to verify that a system possesses the required capability. For example, a capability check would be useful if one would like to know if the entire memory is available, or if all of the processors in a multiprocessor system are working properly. Or, one might want to know if the ALU in the processor is working properly, in which case the capability check would again be used. [Ref. 4]

c. N-Version Programming

N-version programming attempts to tolerate the design faults in software modules. The basic principle of the N-version programming is to produce the code of the same application N-times with the same specifications and same functionality, but by using different programmers and then voting among the outputs of these N results produced by these different software versions. (Illustrated in Figure 2.6.) The concept of

producing different versions of the same software provides different failure modes in each version and is called *design diversity*.

Unfortunately, even different programmers can make the same mistake, or a common mistake can arise from the original specification. *N*-version programming is expensive, raising the system implementations and maintenance cost by factor of *N* or more [Ref. 4].

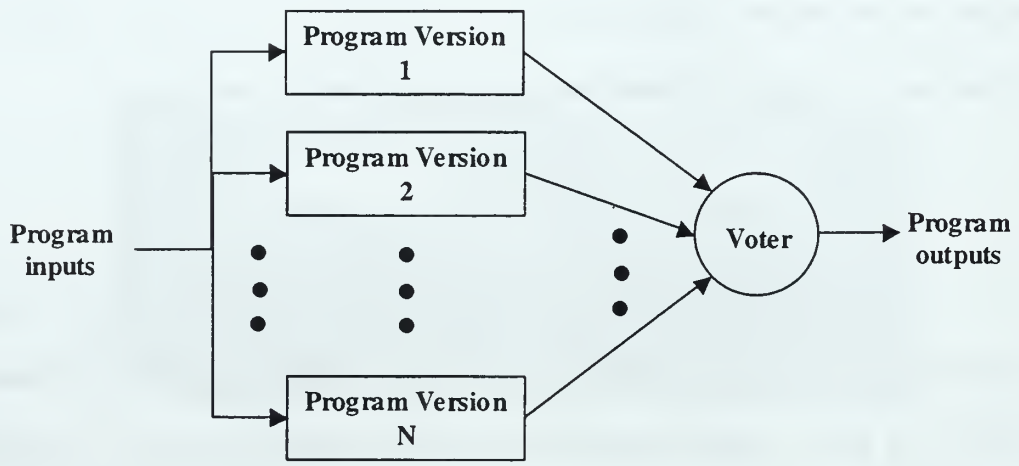


Figure 2.6. The N-Version Programming Concept. [From Ref. 4]

d. Recovery Blocks

The recovery block approach is very similar to the cold standby sparing approach that is used in hardware redundancy. The concept is illustrated in Figure 2.7. One of the *N* versions of a program, called primary version, is always used, unless it fails to pass the acceptance tests. The acceptance checks are, essentially, checks performed on the results produced by the program and may be created using consistency checks and

capability checks. If the primary version fails to pass the acceptance tests, then the first secondary version is tried. This process continues until one version passes the acceptance tests. When all versions fail to pass the acceptance test, a system failure will occur. Assuming perfect coverage and independent faults, the recovery block approach can tolerate up to $N-1$ faults. [Ref. 4]

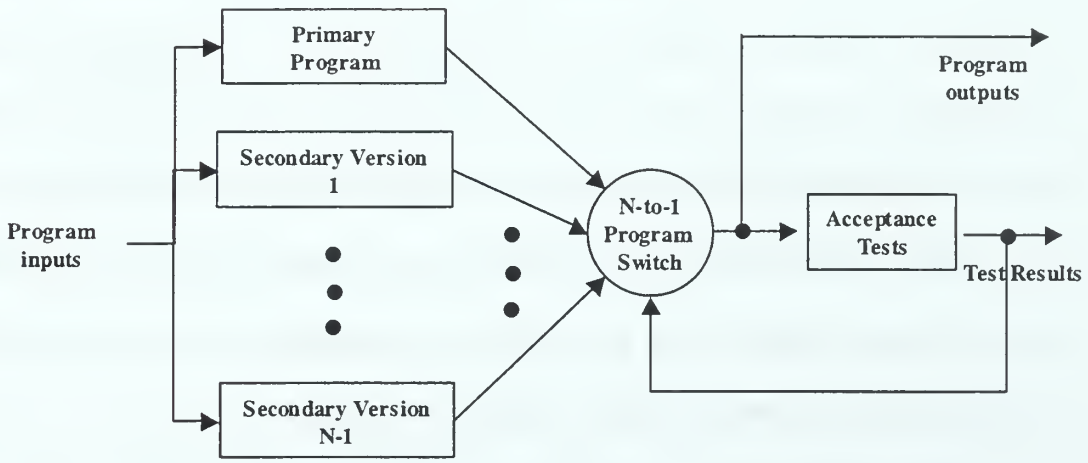


Figure 2.7. The Recovery Block Approach. [From Ref. 4]

3. Information Redundancy

Information redundancy refers to the addition of redundant information to the data with the objective of providing fault tolerance. The purpose of the information redundancy is to protect the state of the information or to protect the transport of messages. The basic idea behind adding extra bits is so that errors in some bits can be detected, and if possible, corrected. The process of adding check bits is called *encoding*. The reverse process of extracting the information from the encoded information is called *decoding*. Error detection and error correction codes (e.g., the Hamming Code) are examples of the information redundancy technique.

4. Time Redundancy

Time redundancy refers to the repetition of a computation or communication action in the domain of time. The purpose of time redundancy is to detect and possibly tolerate the occurrence of transient faults.

The repetition of computation can be used either to compare the results of different computations to determine if a discrepancy exists, or to determine if the existing discrepancy has been corrected or not. This approach is effective when the fault causing the erroneous state is transient. In order to repeat the computation correctly each time, it is essential that the same data is provided to the system. However, when the system enters into an erroneous state, data used in the computation may be corrupted. If this happens, then it becomes impossible to repeat the computation.

The repetition of the communication action can be used to tolerate transient faults resulting in errors in the messages transmitted among the system components. If the message is corrupted or lost due to a transient fault, then repeating the message transmission most likely will not introduce the same error again.

C. OBJECTIVES OF FAULT TOLERANCE

Fault tolerance is an attribute that is designed into a system to achieve some design requirements. The significant requirements are dependability, reliability, availability, safety, performability, maintainability, and testability. Fault tolerance is one system attribute capable of fulfilling such requirements. [Ref. 4]

1. Dependability

Dependability is defined as the quality of the delivered service such that reliance can justifiably be placed on this service [Ref. 3]. Dependability covers all the measures used for quantifying the quality of the delivered service such as reliability, availability, safety, maintainability, and testability.

2. Reliability

The *reliability* of a system is a function of time, $R(t)$, defined as the conditional probability that the system performs correctly throughout the interval of time, $[t_0, t]$, given that the system was performing correctly at a time t_0 [Ref. 4]. In other words, reliability is a measure of the continuous service accomplishment from a reference initial instant. If the lifetime of a system is exponentially distributed, then the reliability of that system is:

$$R(t) = e^{-\lambda t} \tag{2.1}$$

The parameter λ is called the *failure rate* of the system, and is defined as the expected number of failures of a system per unit of time. The commonly accepted relationship between the failure rate and time for electronic components is called the *bathtub curve*, and illustrated in Figure 2.8. The decreasing section of the bathtub curve is called the infant mortality phase. The increasing section of the bathtub curve is called the worn-out phase. In this region, failures begin to appear and increase rapidly due to the physical wearing of electronic components. The intermediate phase is called the useful

phase of the component. During this phase the failure rate is assumed to be constant, which is the λ value explained above.

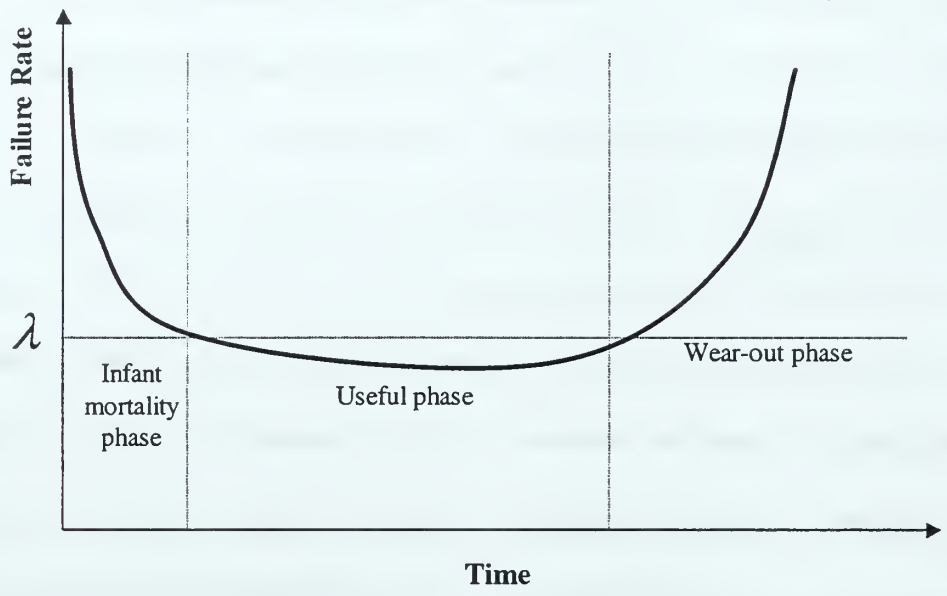


Figure 2.8. Bathtub Curve Relationship Between Failure Rate and Time. [From Ref. 4]

The exponential relationship between the reliability and the time is known as the *exponential failure law*. The exponential failure law is extremely valuable for the analysis of electronic components, and is by far the most commonly used relationship between reliability and time. [Ref. 4]

When the exponential failure law is applied to a system, the life of the system is assumed to be exponentially distributed. With this assumption, the Mean Time to Failure (MMTF) (or expected life) of the system can be calculated with the following equation:

$$MTTF = \frac{1}{\lambda} \tag{2.2}$$

3. Availability

Availability is a function of time, $A(t)$, defined as the probability that a system is operating correctly and is available to perform its functions at the time t. Availability is related to, but different than reliability. Reliability measures how frequently the system fails, whereas availability measures the percentage of time the system is in its operational state. When the mean time to failure of a system is represented as MTTF, and the Mean Time to Repair of the failed system is represented as MTTR, then the availability, α, is calculated as follows:

$$\alpha = \frac{MTTF}{MTTF + MTTR} \tag{2.3}$$

System availability is frequently classified by measuring the percent of time in which the system is available. Table 2.1 shows these common classes and the associated availability percentages and related annual downtime. Systems are characterized as having a certain number of "*9*"s (e.g., "five nines system") or as being a certain availability class (e.g., "Class 5") according to the band of availability it achieves. A Class 5 system, for example, has 99.999% - 99.9999% availability.

AVAILABILITY MEASUREMENT	ANNUAL OUTAGE	AVAILABILITY CLASS	
90%	More than a month	One nines	Class 1
99%	Just under four days	Two nines	Class 2
99.9%	Just under nine hours	Three nines	Class 3
99.99%	About an hour	Four nines	Class 4
99.999%	A little over five minutes	Five nines	Class 5
99.9999%	About half a minute	Six nines	Class 6
99.99999%	About three seconds	Seven nines	

Table 2.1. Availability Classes. [From Ref 10]

4. Safety

Safety, $S(t)$, is the probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the other systems or compromise the safety of any people associated with the system. Safety is a measure of the fail-safe capability of the system; if the system does not operate correctly, it is desirable to have the system fail in a safe manner. [Ref. 4]

In order to increase the safety of a system, the likelihood of undetected error in the output should be made negligible so that when an uncorrectable error in the output is detected, it is possible to carry out the safe failure of the system.

Although the concept of safety seems similar to that of reliability, they are in fact different. Reliability is the probability that a system will perform its functions correctly, whereas safety is the probability that a system will either perform its functions correctly

or will discontinue the functions in a harmless manner. It may be noted that when a system is reliable, it is also safe. However the reverse is not always true.

5. Performability

The *performability* of a system is a function of time, $P(L,t)$, defined as the probability that the system performance will be at or above, some level, L, at the instance of time, t. Performability differs from reliability in that reliability is a measure of the likelihood that all of the functions are performed correctly, while performability is a measure of the likelihood that some subset of functions is performed correctly. [Ref. 4]

Graceful degradation, which refers to the ability of a system to automatically decrease its performance level, is an important system feature closely related to performability. Fault tolerance can support graceful degradation and performability by neutralizing the effects of hardware and software faults of a system, thereby allowing performance at some reduced level. [Ref. 4]

6. Maintainability

The *maintainability* is a measure of the ease with which a system can be repaired once it has failed. In other words, maintainability is the probability, $M(t)$, that a failed system will be restored to an operational state within a period of time, t. [Ref. 4]

Maintainability encapsulates not only the failures of the system, but also the modifications that are necessary for the required level of system performance. In order to keep a system in a state that is relevant to its users, it is mandatory to repeatedly modify and enhance the system functions. The ease with which such modifications can be

performed is dependent on the modularization of the system. If the consequences of a modification can be localized to well-defined small modules, then the maintenance effort can be minimized.

Fault tolerance can support maintainability in the problem detection and problem location process. Once the problem is detected and located maintenance can be performed. Fault tolerance can also support maintainability in the modification process by allowing maintenance actions without interrupting the service delivered by the system.

7. Testability

The *testability* is simply the ability to test for certain specifications of the system. In order to improve the testability, certain tests can be automated and integrated into the system. Fault tolerance techniques can be used to detect and locate the problems for the purpose of improving testability. [Ref. 4] Since a system must be retested after every modification or fix, testability is closely related to maintainability.

D. PHASES IN FAULT TOLERANCE

The implementation of fault tolerance in a particular system is dependent upon the system itself, and differs from one system to another. Every system requires a different type of implementation of fault tolerance depending on its needs, functionality, and architecture. Therefore it is very difficult to propose a general technique for adding fault tolerance to a system. However, there are some general actions that systems must perform during the implementation of fault tolerance. These actions can be classified

according to the phase in which they occur: *error detection*, *damage confinement*, *error correction*, and *fault treatment and continued system service*.

1. Error Detection

Before starting any fault tolerance activity, an error must be detected. Presence of the fault and failure cannot be observed directly. Since the error is defined as a state of a system, the presence of error can be detected by checking the system's states. Afterwards, the presence of failures and faults can be deduced from the detected error in the system. The effectiveness of the fault tolerance implementation depends directly on the effectiveness of the error detection mechanism employed.

There are some important properties that an ideal error detection mechanism should satisfy. First, an ideal check should be determined solely from the specifications of the system, and should not be influenced by the internal design of the system. Any influence of the system on the check can cause the same error in the check as is present in the system. For that reason, while designing an error detection mechanism, the system should be treated as a "black box". [Ref. 1]

Secondly, the error detection mechanism should be complete and correct. All possible errors should be detected, and all declared errors should actually be present in the system. In other words, the detection mechanism should prevent false alarms.

Thirdly, the check should be independent from the system with respect to susceptibility of faults. If the detection mechanism fails when the system fails, then the check is useless. The detection mechanism should have a different failure mode than the

system. This minimizes the probability that the detection mechanism will fail at the same time as the system.

Implementation of real detection mechanisms rarely satisfies all criteria explained above, due to their complexity, impracticality or cost. Therefore, instead of ideal checks, acceptable checks are used for error detection in real designs. An acceptable check is an approximation performed by ignoring errors that rarely occur. This type of checking design aims to lower the cost of implementation, and at the same time tries to maximize errors detected.

In computer systems, different types of error detection mechanisms are employed depending on the system and the errors to be detected. In the following sections, some general types of checks that are most frequently employed in computer systems will be discussed.

a. Replication Checks

In this type of check, some components of the system are replicated as many times as needed depending upon the application, and then the results of these components are compared or voted to detect the errors. Since replication checks involve replication of the system components, it is one of the most expensive methods of error detection. However, this type of check can be fairly complete and can be implemented without the internal design information of the system being replicated.

b. Timing Checks

Timing checks are used for controlling the time-related constraints of the system in order to see if those constraints are being met. Usually a timer is set to a value according to the system's specifications. Expiration of the timer indicates that the time-related constraints of the system are violated. Timing checks are used in both hardware and software systems.

Timing-related errors are very important, especially in distributed systems. In most distributed systems, a working node must respond within some pre-determined time to show that it is up and running. If a node fails to respond within the timeout period, then its failure is declared. This is the most common method of detecting node failures.

c. Structural Checks

When data is the primary concern of the fault tolerance, structural checks are used to detect errors. In structural checks, redundancy of information added to the data to be protected can be used for detecting the errors.

Structural checks are mostly used in hardware in a process called *coding*. In the coding mechanism, some extra bits are added to the actual data bits. These coding bits are calculated according to relationship rules based on the value of data bits. The structural check mechanism recalculates coding bits and compares them with the existing ones. When the coding bits or the data bits are corrupted, newly calculated coding bits will not match the old ones, and thus the error will be detected (e.g., digital signatures.)

d. Reasonableness Checks

Reasonableness checks determine if the state of some object in the system is "reasonable." Reasonableness checks can be implemented either by checking the range or the rate of change. The range checks attempt to determine if a certain value is within a specified range. The rate of change checks attempt to determine if the rate of change of a certain value is within specified bounds.

e. Diagnostics Checks

Diagnostic checks are implemented by employing special input values to a system, whose output values are known. These types of detection mechanisms are usually built into the system and are used for the system's initial self-checking process.

2. Damage Confinement and Assessment

The main goal of the damage confinement and assessment phase is to determine the boundaries of corruption before starting the error recovery process. During the time delay between the failure and the event of error detection, an error may propagate and spread to other parts of the system.

The main reason for the error propagation is that the communication takes places among the system components. For that reason, the information flow between the components of the system has to be examined after the error is detected in order to assess the extent of the damage. The goal is to identify a state in which no information exchange has occurred. Then the damage could be limited to this boundary.

3. Error Correction

After the error is detected and the damage is assessed, the erroneous state of the system should be corrected. This correction can be made using a process called *effective error processing*. Effective error processing refers to the correction made after an error has taken effect. The goal of effective error processing is to bring the effective error back to a latent state, and before occurrence of failure if possible. Effective error processing may take two forms: *error recovery* and *error masking*.

a. Error Recovery

The error recovery mechanism typically denies the service request and sets the system to an error-free state so that it can service subsequent requests. Error recovery can be classified as *backward error recovery* and *forward error recovery*.

In backward error recovery, when the error is detected, the system is restored to previously occupied (correct) state prior to the error becoming effective. In this method, states of the system are periodically checkpointed on some stable storage that would not be affected by a failure. When the error is detected, the system is rolled back to the last checkpointed state, which is assumed to be error free. It is very suitable for transient faults, because restarting the system from the last checkpointed state will not introduce the error again. Since checkpointing has to be performed periodically on a stable storage, the backward error recovery technique introduces a great amount of overhead to the system. However, due to its simplicity, the backward error recovery mechanism is the most commonly used error recovery technique.

In forward error recovery, instead of rolling back, the system is set to a new error-free state (one not previously occupied) by taking the necessary corrective actions. In order to decide on the necessary actions, the exact nature of the error has to be known, and the exact damage has to be determined. These qualifying characteristics make the forward error recovery technique very difficult to implement.

b. Error Masking

In error masking, the erroneous state of the system contains enough redundancy to enable the delivery of an error-free service from the erroneous internal state. Examples of error masking are the error-correcting codes used for electronic, magnetic, and optical storage. Additionally, NMR technique, discussed previously in the passive hardware redundancy section, can be given as an example of error masking.

4. Fault Treatment and Continued Service

Unlike the first three phases, this phase does not deal with errors. Faults are the main focus of the fault treatment and continued phase. If the fault is transient, then when the system is restarted from the error free state, the same problem will not occur again. However, if the error is caused by a permanent fault, then restarting the system from the error-free state will cause the same error again. Thus, the identification of the faulty component and its exclusion from the computation performed after recovery is essential. This phase can be divided into two phases. These sub-phases are known as the fault location phase and the system repair phase.

In the fault location phase, the component of the system containing the fault is identified. In the system repair phase, the located faulty component is repaired. This repair can be done on-line and without manual intervention.

When the system repair phase is completed, the system can continue to provide its services again. The overall effect of fault tolerance phases on the system would take the form of a minor discontinuity in service or some performance degradation.

THIS PAGE INTENTIONALLY LEFT BLANK

III. FAULT TOLERANCE IN WINDOWS NT OPERATING SYSTEM

The SAAM server runs as an application and builds a Path Information Base (PIB) on the volatile memory to support QoS routing. Specifically, the server identifies those paths or sub paths that can potentially be used to route flows, and maintains up-to-date performance parameters for each of them. The server computes path performance parameters by aggregating link level performance data passed up from each router.

The SAAM server is currently prototyped as a Java application on the Windows NT Server 4.0 operating system environment. When choosing Windows NT Server, influencing factors included reliability, scalability, stability, speed, and ease of administration.

Since the SAAM project is currently prototyped in the Windows NT operating system environment, it is essential to be aware of the provided fault tolerance related features with this operating system to not reinvent the wheel. Therefore, this chapter will focus on the Windows NT server operating system features that support fault tolerance.

The Windows NT Server 4.0 includes the following fault tolerance related features:

- Error handling and protected subsystems
- NT File System
- Automatic restart
- Tape backup support
- Uninterruptible power supply (UPS) support
- Fault Tolerant Storage

In addition to these features, the Windows NT Server Enterprise Edition offers the following services:

- Microsoft Cluster Server (MSCS)
- Windows NT Load Balancing Service (WLBS)

A. ERROR HANDLING AND PROTECTED SUBSYSTEMS

Software applications do not always operate as expected; they can enter into faulty states. Windows NT Server is designed to tolerate these faults by ensuring that they do not affect other components of the operating system. For the Windows NT Server, the first line of defense against software errors is its structured method of exception handling. When an abnormal event occurs, the event is captured and either the processor or the operating system issues an exception. This design ensures that no undetected error is allowed to influence the system or other user programs. [Ref. 5]

Windows NT Server also employs protected subsystems in its design. Protected subsystems are separate, unique memory locations that are assigned to different processes and applications. By isolating programs in this way, Windows NT Server ensures that a program fault will not affect the system's kernel and, as a result, crash the operating system. Similarly, programs are isolated from each other so that when a program faults, it does not adversely affect other programs running on the system. This architecture makes it safe to deploy new Windows NT Server-based applications. New applications can be run and tested on a Windows NT Server-based machine without concern that they will adversely affect the system or other production applications. As a result, deploying

powerful, new server-based applications on Windows NT Server is less risky than it is with some other server operating systems. [Ref. 5]

B. NT FILE SYSTEM (NTFS)

NTFS is a recoverable file system that uses caching and allows recovery from a disk failure. NTFS helps guarantee the consistency of the disk volume by using standard transaction logging and recovery techniques, although it does not guarantee the protection of user data. All data is accessed via the file cache. While the user searches folders or reads files, data to be written to disk accumulates in the file cache. If the same data is modified several times, all those modifications are captured in the file cache. The result is that the file system needs to write to a disk only once to update the data. [Ref. 6]

When a disk error occurs during a write operation, NTFS is capable of automatically re-mapping the bad sector, and allocates a new cluster for the data. The following section discusses how Windows NT automatically recovers from some kinds of disk errors. Windows NT provides two kinds of disk error recovery: *dynamic data recovery by using sector sparing* and *NTFS cluster remapping*.

Dynamic data recovery by using sector sparing is only available on SCSI disks that are configured as part of a fault-tolerant volume. Sector sparing works on fault-tolerant volumes because a copy of the data on the sector with the error can be regenerated. Windows NT Server obtains a spare sector from the disk device driver to replace the bad sector. It then recovers the data by reading the sector from the mirror disk or recalculating the data from a stripe set with parity, and writes the data to the new sector. This processing is transparent to any applications performing disk input/output

(I/O) operation. Sector sparing eliminates error messages such as the "Abort, Retry, or Fail?" that occur when a file system encounters a bad sector. [Ref. 7]

NTFS cluster remapping is another disk recovery technique. When Windows NT returns a bad sector error to the NTFS file system, NTFS dynamically replaces the cluster containing the bad sector and allocates a new cluster for the data. If the error occurred during a read on a non-fault-tolerant volume, NTFS returns a read error to the calling program, and the data are lost. When the error occurs during a write, NTFS writes the data to the new cluster, and no data are lost. NTFS puts the address of the cluster containing the bad sector in its Bad Cluster File so the bad sector will not be reused. [Ref. 7]

Windows NT Server provides additional recovery mechanisms for fault-tolerant volumes (mirror sets and stripe sets with parity). Table 3.1 summarizes what happens if a sector goes bad on a computer running Windows NT Server.

C. AUTOMATIC RESTART

The Windows NT operating system includes an automatic restart feature. In the event of a failure, the system can be configured to automatically restart itself. This feature of Windows NT Server provides maximum system up-time. To assist the administrator in determining the cause of the failure, Windows NT Server can be set to transfer its memory contents to a disk file before restarting for later analysis. [Ref. 5]

Description	Fault-Tolerant Disk (FtDisk) installed with a SCSI disk that has spare sectors	Fault-Tolerant Disk (FtDisk) installed with a non-SCSI disk or disk with no spare sectors	Fault-Tolerant Disk (FtDisk) not installed with any kind of disk
Fault-tolerant volume (Windows NT Server only)	1. FtDisk recovers the data. 2. FtDisk replaces the bad sector. 3. File system does not know about the error.	1. FtDisk recovers the data. 2. FtDisk sends the data and bad-sector error to the file system. 3. NTFS performs cluster remapping. 4. FAT doesn't do anything about the error.	N/A
Non-fault-tolerant volume	1. FtDisk cannot recover the data. 2. FtDisk sends a bad-sector error to the file system. 3. NTFS performs cluster remapping. On a read operation, data are lost. 4. FAT loses the data on both read and write.	1. FtDisk cannot recover the data. 2. FtDisk sends a bad-sector error to the file system. 3. NTFS performs cluster remapping. On a read operation, data are lost. 4. FAT loses the data on both read and write.	1. Disk driver returns a bad-sector error to the file system. 2. NTFS performs cluster remapping. On a read operation, data are lost. 3. FAT loses the data on both read and write.

Table 3.1. What Happens If a Sector Goes Bad? [From Ref. 8]

D. TAPE BACKUP SUPPORT

Regular tape backup is an important part of guaranteeing data availability. Windows NT Server includes a graphical tool called *Backup* that makes it easy to backup Windows NT Server-based data to tape. Windows NT Backup provides five backup types: *normal*, *copy*, *incremental*, *differential*, and *daily copy*. Some backup types use backup markers, also known as archive attributes, to track when a file has been backed up. Table 3.2 describes the backup types.

Backup Type	Actions
Normal (Full)	Backs up selected files and marks each as having been backed up. With normal backups, one can restore files quickly because files on the last tape are the most current.
Incremental	Backs up only those files created or changed since the last normal or incremental backup. It marks files as having been backed up.
Differential	Backs up those files created or changed since the last normal (or incremental) backup. It does not mark files having been backed up.
Copy	Backs up selected files, but does not mark each file as having been backed up. Copying is useful if user wants to back up files between normal and incremental backups, because copying does not invalidate these other backup operations.
Daily copy	Backs up selected files that have been modified the day the daily backup is performed. The backed up files aren't marked as having been backed up.

Table 3.2. Backup Types and Their Functions. [From Ref. 9]

E. UNINTERRUPTIBLE POWER SUPPLY (UPS)

The Uninterruptible Power Supply (UPS) service is a system software component of Windows NT, which can be configured to detect and warn of impending power failure. It has built-in electronics that constantly monitor line voltages. If the line voltage fluctuates above or below pre-set limits, or fails entirely, the UPS supplies power to the computer system from built-in batteries. UPS Systems provide a hardware interface that can be connected to the computer. Using appropriate software, this interface enables an orderly handling of the power failure, including performing a system shutdown before the UPS batteries are depleted.

UPS offers significant benefits when considering the fact that power loss accounts for almost 27% of all unplanned outages. In some locations, and at certain times of the year, power outages can occur as often as once a day. Operators should use redundant power supplies for maximum reliability.

Windows NT has built-in UPS functionality that takes advantage of the special features that many UPS systems provide. These features ensure the integrity of data on the system and allow the computer system and UPS to be shutdown in a controlled manner if a power failure outlasts UPS batteries. In addition, users connected to a computer running Windows NT Server can be notified that a shutdown will occur and new users are prevented from connecting to the computer. Finally, damage to the hardware from a sudden, uncontrolled shutdown can be prevented. [Ref. 6]

F. FAULT-TOLERANT STORAGE

The term, Redundant Array of Inexpensive Disks (RAID), was first coined by Chen, Gibson, Katz, and Patterson of the University of California at Berkeley. [Ref. 16] The RAID Advisory Board (RAB) has since re-named the term, replacing "inexpensive" with "independent."

RAID minimizes loss of data caused by problems with accessing data on a hard disk. RAID is a fault-tolerant disk configuration in which part of the physical storage capacity contains redundant information about data stored on the disks. The redundant information enables regeneration of the data if one of the disks or the access path to it fails, or a sector on the disk cannot be read. [Ref. 7]

Some vendors sell disk subsystems that implement RAID technology completely within the hardware. Some of these hardware implementations support hot swapping of disks, which enables the user to replace a failed disk while the computer is still running Windows NT Server. [Ref. 7] Regardless of their implementation techniques, all RAID disk configurations perform the following functions:

- *Regeneration* of data to satisfy a read request when a disk or a path to a disk has failed.
- *Reconstruction* of the missing data onto the new disk when the user has replaced the failed disk (or the path to it).

Normally a RAID set appears as a single large disk drive to applications and the operating system, although it is actually an array of drives with equal capacity. RAID terminology is standardized by level, as indicated in Table 3.3.

RAID Level	Functionality
0	Data are stripped across available disk drives, to improve access times and throughput. There is no redundancy.
1	Two disk drives are mirrored (both store the same data), using a single disk controller. Data can be read off both drives simultaneously (either drive can service any request), providing improved performance for reads (but not for writes), and redundancy.
2	Data are spanned (stripped, bit-by-bit) across multiple disks, and additional disks are used to store Hamming codes (to detect and correct errors or recover from failed drives). Four data disks would require three additional error detection and correction disks.
3	Data are stripped (sometimes called *interleaved*) either bit-by-bit or (more commonly) byte-by-byte across two or more (four is apparently best) data disks (for example, first byte to first disk, second byte to next disk, and so on –written in parallel to all disks). A parity byte is constructed from the corresponding bytes on the data disks and is written to one additional disk, which is dedicated as a *parity disk*. The contents of a failed disk can be reconstructed from the other disks. However, the use of a single parity disk creates a performance bottleneck.
4	Same as RAID 3, but data are stripped (and parity is constructed) in disk sectors (which is the smallest unit of disk storage allocation) rather than bits or bytes.
5	Data are stripped sector by sector across two or more disks. Parity information sectors are stripped along with the data on each disk, and there is no dedicated parity disk. Since both parity and data are stripped, simultaneous writes are possible (depending on where the data has to go).

Table 3.3. RAID Levels.

Windows NT Server provides a software implementation of disk striping at RAID level 0 and disk mirroring at RAID level 1. It also provides an implementation of RAID level 5. Cluster server services in the Windows NT Server Enterprise Edition uses RAID subsystems exclusively.

1. Stripe Set

Stripe sets are composed of stripes of equal size on each disk in the volume. One can create a stripe set from equal sized, unallocated areas on two to 32 physical disks. For Windows NT Workstation and Windows NT Server, the size of the stripe is 64K.

Stripe sets do not contain any redundant information. Therefore, the cost per MB for a stripe set is identical to that of the same amount of storage configured from a contiguous area on a single disk. Although the data are spread across multiple disks, there is no fault tolerance. When any disk fails, the whole stripe set fails, and no data can be recovered. The reliability for the stripe set is no better than the least reliable disk in the set. [Ref. 7]

A stripe set may be used for performance reasons. Access to the data on a stripe set is usually faster than access to the same data on a single disk, because the I/O is spread across more than one disk. Therefore, Windows NT can perform doing seeks on more than one disk at the same time, and can even have simultaneous reads or writes occurring. [Ref. 7]

2. Mirror Set

A mirror set provides an identical twin for the selected partition. All data written to the mirror set are written to both partitions, which results in disk space utilization of only 50 percent. Creating a mirror set is similar to making a copy of a document by using a copy machine. The original partition is like the original of the document, and the shadow partition is the copy. Unlike a copy machine, however, Windows NT continually updates both the original and shadow partitions when any changes are made to the mirror

set. It is not necessary to use identical physical disks or to have the same partitions on each disk, although identical disks should be used if putting the system partition on a mirror set. A mirror set requires only sufficient unused space on the second disk to create the shadow partition. [Ref. 7]

If there is a read failure on one of the disks, the fault-tolerant disk driver reads the data from the other disk in the mirror set. If there is a write failure on one of the disks in the mirror set, the fault-tolerant disk driver uses the remaining disk for all accesses. Because dual-write operations can degrade system performance, many mirror set implementations use duplexing, where each disk in the mirror set has its own disk controller. [Ref. 7]

3. Stripe Set With Parity

The parity strip is the exclusive-OR (XOR) of all the data values for the data strips in the stripe. If no disks in the stripe set with parity have failed, the new parity for a write can be calculated without having to read the corresponding strips from the other data disks. Thus, only two disks are involved in a write operation, the target data disk and the disk that contains the parity strip. Figure 3.1 shows the steps that are involved in writing data to a stripe set with parity. [Ref. 7]

When implementing a stripe set with parity, there must be at least three disks and no more than 32 disks in the set. The physical disks do not need to be identical. However, there must be equal size blocks of unpartitioned space available on each physical disk in the set. The disks can be on the same or different controllers. As with stripe sets, one

cannot add disks to a stripe set with parity if one may need to increase the size of the volume later. [Ref. 7]

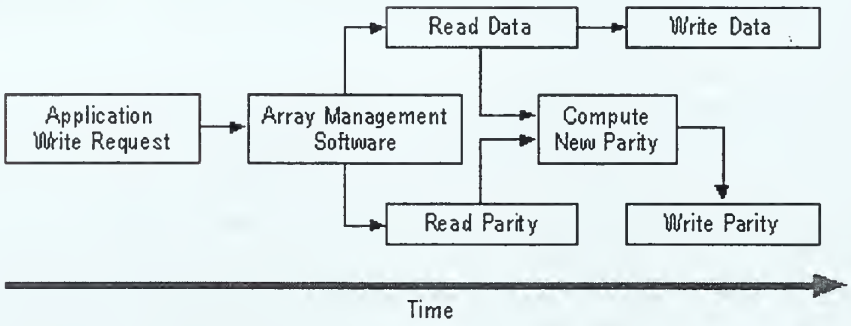


Figure 3.1. Steps in Writing Data to a Stripe Set with Parity.

If one of the disks in a stripe set with parity fails, none of the data are lost. When a read operation requires data from the failed disk, the system reads all of the remaining good data strips in the stripe and the parity strip. Each data strip is subtracted (with XOR) from the parity strip. The result is the missing data strip. [Ref. 7]

When the system needs to write a data strip to a disk that has failed, the system reads the other data strips and the parity strip and then backs them out of the parity strip, leaving the missing data strip. The modifications needed to the parity strip can now be calculated and made. Only the parity strip is written upon; the data strip is not written upon because it is bad.

There is no effect on a read operation when the failed disk contains a parity strip. (The parity strip is not needed for a read unless there is a failure in a data strip.) When the failed disk contains a parity strip, the system does not compute or write the parity strip when there is a change in a data strip. [Ref. 7]

G. MICROSOFT CLUSTER SERVER (MSCS)

1. Overview of Server Cluster

Clusters of computer systems have been built and used for over a decade. Pfister defines a cluster as "a parallel or distributed system that consists of a collection of interconnected whole computers, that is utilized as a single, unified computing resource." [Ref. 10] The goal of a cluster is to make it possible to share a computing load over several systems on a network without either the users or system administrators needing to know that more than one system is involved.

In a cluster, if a certain resource or set of resources goes down, the system intelligently chooses where and how to run applications in the network. With clustering, one of two nodes can also be used to run certain services while the other is used for maintenance. Later, the maintained node can be returned to the cluster without affecting services. In short, clustering provides the high availability of a multiple-node network with the management simplicity of a single address space. [Ref. 12]

There are basically three techniques that clusters use to make disk data available to more than one server:

- **Shared disks**: In the shared disk model, software running on any system in the cluster may access any resource (e.g., a disk) connected to any system in the cluster. If two systems need to see the same data, the data must either be read twice from the disk or copied from one system to another. [Ref. 11]

- **Mirrored disks**: A more flexible alternative is to let each server have its own disks, and to run software that "mirrors" every write from one server to a copy of the data on at least one other server. This technique can be used for keeping the data at a disaster recovery site in synch with that of a primary server.
- **Shared nothing**: In the shared nothing software model, each system within the cluster owns a subset of the cluster's resources. Only one system may own and access a particular resource at a time. However, upon failure, another dynamically determined system may take ownership of the resource of the failed system. In addition, requests from clients are automatically routed to the system that owns the resource. For example, if a client request requires access to resources owned by multiple systems, one system is chosen to host the request. The host system analyzes the client request and ships subrequests to the appropriate systems. Each system executes the sub-request and returns only the required response to the host system. The host system assembles a final response and sends it to the client. [Ref. 14]

2. MSCS

MSCS, also known as "Wolfpack", is a built-in feature of the Windows NT Server, Enterprise Edition. It is software that supports the connection of two servers into a "cluster" for higher availability and easier manageability of data and applications. MSCS can automatically detect and recover from server or application failures. It can be used to move server workload to balance utilization and to provide for planned maintenance without downtime. [Ref. 13]

MSCS uses software "heartbeats" to detect failed applications or servers. In the event of a server failure, it employs a "shared nothing" clustering architecture that automatically transfers ownership of resources (such as disk drives and IP addresses) from a failed server to a surviving one. It then restarts the failed server's workload on the surviving server. If an individual application fails (but the server does not), MSCS will typically try to restart the application on the same server. If that fails, the MSCS moves the application's resources and restarts the same application on the other server. The cluster administrator can use a graphical console to set various recovery policies, such as dependencies between applications, whether or not to restart an application on the same server, and whether or not to automatically "failback7" (rebalance) workloads when a failed server comes back online. Generic MSCS architecture is shown in Figure 3.2.

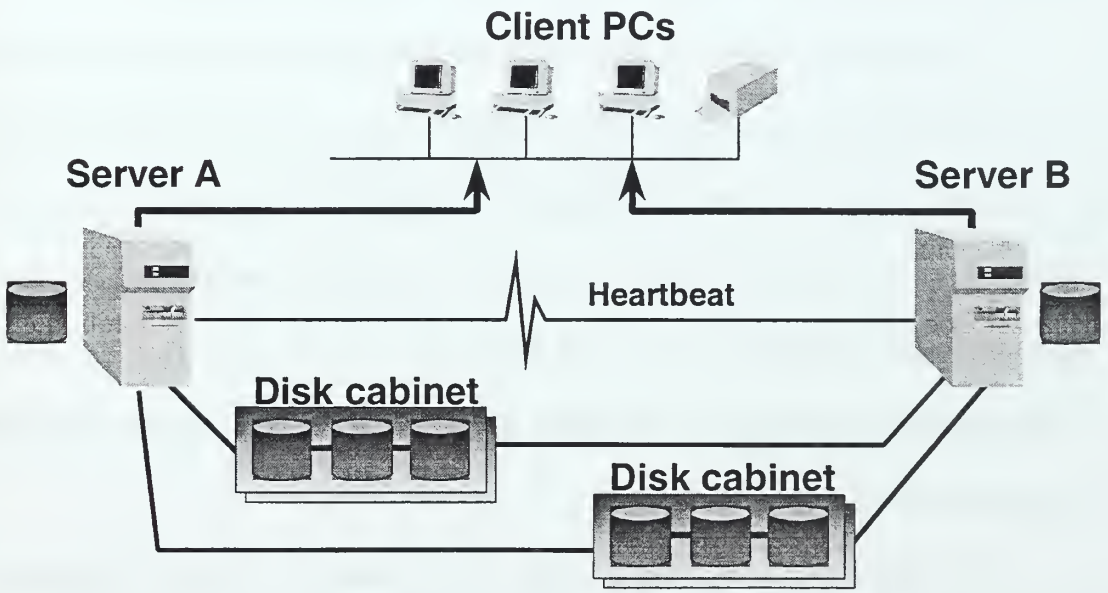


Figure 3.2. A Generic MSCS Setup. [After Ref. 12]

Figure 3.3 shows an overview of the components and their relationships in a single system of a Windows NT cluster. Microsoft Cluster Server mainly comprised of three key components:

- The Cluster Service
- The Resource Monitor
- Resource and Cluster Administrator Extension DLLs

The Cluster Service (which is composed of the Event Processor, the Failover Manager/Resource Manager, the Global Update Manager, the Communication Manager, the Checkpoint Manager, and Membership Manager) is the core component of MSCS and runs as a high-priority system service. The Cluster Service controls cluster activities and performs such tasks as coordinating event notification, facilitating communication between cluster components, handling failover operations, and managing the configuration. Each cluster node runs its own Cluster Service. [Ref. 32]

The Resource Monitor is an interface between the Cluster Service and the cluster resources, and runs as an independent process. The Cluster Service uses the Resource Monitor to communicate with the resource DLLs. The DLL handles all communication with the resource, thus shielding the Cluster Service from resources that misbehave or stop functioning. Multiple copies of the Resource Monitor can be running on a single node, thereby providing a means by which unpredictable resources can be isolated from other resources. [Ref. 32]

The Resource Monitor and resource DLL communicate using the Resource API, which is a collection of entry points, callback functions, and related structures and macros used to manage resources. Applications that implement their own resource DLLs

to communicate with the Cluster Service and that use the Cluster API to request and update cluster information are defined as cluster-aware applications. Applications and services that do not use the Cluster or Resource APIs and cluster control code functions are unaware of clustering and have no knowledge that MSCS is running. These cluster-unaware applications are generally managed as generic applications or services. [Ref. 32]

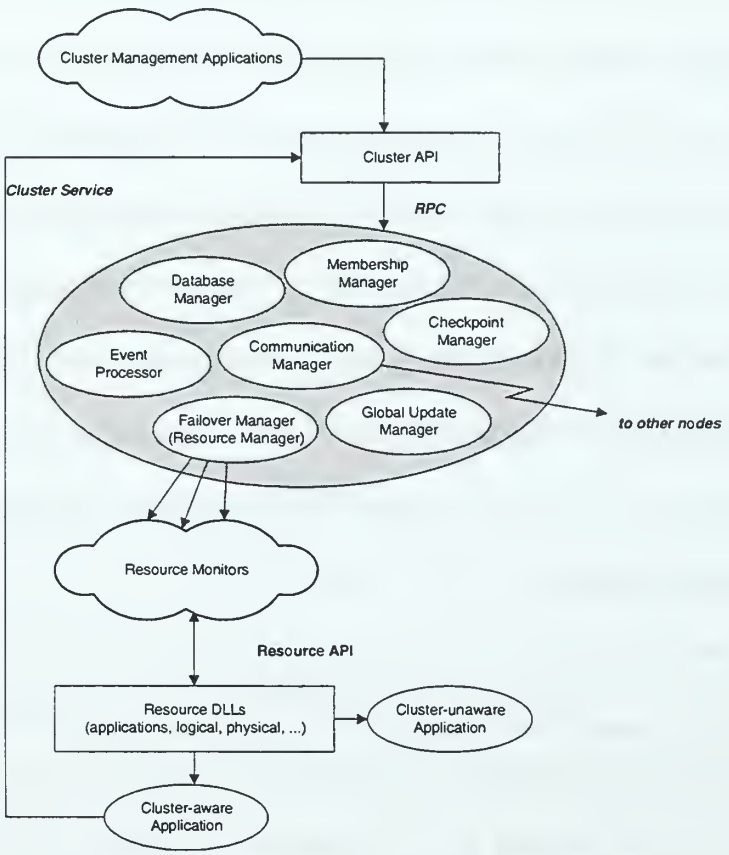


Figure 3.3. MSCS Components on a Single Windows NT Server. [From Ref. 32]

MSCS can reduce planned and unplanned downtime. However, even with MSCS, a server could still experience downtime from the following events:

- MSCS failover time: If MSCS recovers from a server or application failure, or if it is used to move applications from one server to another, the application(s) will be unavailable for a non-zero period of time.
- Failures from which MSCS cannot recover: There are types of failure that MSCS does not protect against, such as loss of a disk not protected by RAID, loss of power when a UPS is not used, or loss of a site when there's no fast-recovery disaster recovery plan. Most of these can be survived with minimal downtime, however, if precautions are taken in advance.
- Server maintenance that requires downtime: MSCS can keep applications and data online through many types of server maintenance, but not in all circumstances. For example, two such circumstances occur when completely upgrading both servers in a cluster, or installing a new version of an application which has a new on-disk data format that requires reformatting preexisting data).

MSCS does not require any special software on client computers, so the user's experience during failover depends on the nature of the client side of their client-server application. Client reconnection is often transparent, because MSCS has restarted the applications, file shares, etc., at exactly the same IP address. [Ref. 13]

If a client is using "state-less" connections such as a standard browser connection, then client would be unaware of a failover if it occurred between server requests. For client-side applications that have "state-full" connections to the server, a new logon is

typically required following a server failure. In many cases, this approach is required for security purposes. [Ref. 13]

The servers in an MSCS cluster cannot be located at separate locations for recovery from site disasters. All of the cluster configurations currently being considered for validation use SCSI connections to storage resources, which limits the distance between clustered servers to the distance supported by standard SCSI. This is typically no more than 25 meters. [Ref. 13] There are three types of server applications that will benefit from MSCS clusters:

1. *"In the box" services of Windows NT Server, Enterprise Edition*: File shares, print queues, Internet/intranet sites managed by the Microsoft Internet Information Server, Windows NT Server's built-in Web server; Microsoft Message Queue Server (MSMQ) services, and Microsoft Transaction Server (MTS) services, both of which are part of Windows NT Server.
2. *Generic applications*: MSCS includes a point-and-click wizard for setting up any well-behaved server application for basic error detection, automatic recovery, and operator-initiated management. A "well behaved" server application is one that keeps a recoverable state on shared SCSI disk(s), and whose client can gracefully handle a pause in service of up to a minute as the application is automatically restarted by MSCS.
3. *Cluster-aware applications*: Software vendors will test and support their application products on MSCS. Over time, vendors will provide MSCS-based enhancements, from simpler setup and faster failover to cluster-enabled scalability and load balancing.

H. WINDOWS NT LOAD BALANCING SERVICE (WLBS)

Microsoft Windows NT Load Balancing Service (WLBS), a feature of Windows NT Server 4.0, Enterprise Edition, provides scalability and high availability to enterprise-wide Transmission Control Protocol/Internet Protocol (TCP/IP) services, such as Web, proxy, Virtual Private Networking (VPN), and streaming media services. WLBS is based on the Convoy Cluster Software by Valence Research, Inc., a recent Microsoft acquisition. [Ref. 15]

The two principal goals of Microsoft Windows NT Load Balancing Service (WLBS) are to provide high availability for Internet server programs and to ensure scale server performance. It accomplishes these goals by using a cluster of two or more computers (called hosts) working together, as shown in Figure 3.4. WLBS installs as a standard Windows NT networking driver and runs on an organization's existing LAN. All servers within a cluster are placed on a single subnet. Internet clients access the cluster using a single IP address (or a set of addresses for a multi-homed host). The clients cannot distinguish the cluster from a single server. [Ref. 17]

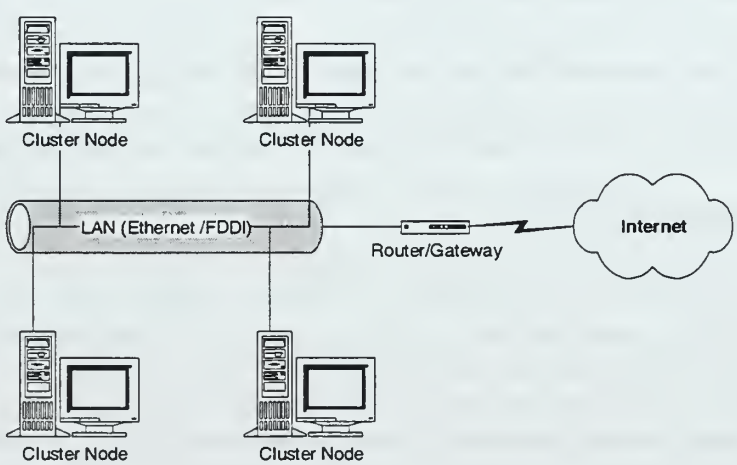


Figure 3.4. An Example Configuration of WLBS. [From Ref. 15]

1. WLBS Features

WLBS cluster servers emit a "heartbeat" to other nodes in the cluster, and listens for the heartbeat of other nodes. When a node fails or goes offline, WLBS automatically reconfigures the cluster to direct client requests to the remaining computers. In addition, for load-balanced ports, the load is automatically redistributed among the computers still operating, and ports with a single server have their traffic redirected to a specific host. While connections to the failed or offline server are lost, the offline computer can transparently rejoin the cluster and regain its share of the workload once the necessary maintenance is completed. In addition, WLBS handles inadvertent subnetting and rejoining of the cluster network. [Ref. 15]

WLBS load-balances incoming TCP/IP traffic across all the hosts in a cluster to scale performance. Running a copy of the server program on each load-balanced host enables the load to be partitioned among them in any manner. WLBS transparently distributes the client requests among the hosts and lets the clients access the cluster using one or more "virtual" IP addresses. Up to 32 hosts may operate in each cluster, and hosts may be added transparently to a cluster to handle increased load. WLBS can also direct all traffic to a designated single host, called the default host.

Load balancing can be specified for a single IP port or group of ports using port management rules that tailor the workload for each service. In addition, optional support for client sessions can be enabled, as well as optional port rules to let all client requests be directed to a single host to further refine load balancing among different applications. Undesired network access can also be blocked to certain IP ports. WLBS logs all actions and cluster changes to the Windows NT event log. [Ref. 15]

Administrators can remotely control WLBS actions from any networked Windows NT-based computer using console commands or scripts. All cluster hosts can be controlled with one command, or controlled individually. The control program has fully encrypted password protection to prevent unauthorized access. [Ref. 15]

No special hardware is needed to interconnect cluster hosts; the cluster may exchange status messages over a single local area network using Ethernet (10, 100, or gigabit) or FDDI adapter cards. WLBS also lets clients access the cluster with a single Internet logical name and IP address, while retaining individual names for each computer. In addition, server applications need not be modified to run in a WLBS cluster, and all operations, including recovery, require no human intervention. Computers can also be taken offline for preventive maintenance without disturbing cluster operations. [Ref. 15]

WLBS supports state-full client sessions and Secure Sockets Layer (SSL). If a server application (such as a Web server) maintains state information about a client session that spans multiple TCP connections, it is important that all TCP connections for this client be directed to the same cluster host. Should a server or network failure occur during a "stateful" client session, a new logon may be required to re-authenticate the client and re-establish session state. WLBS also allows modification of session support to direct all client requests from an IP Class C address range to a single cluster host. This feature ensures that clients who use multiple proxy servers to access the cluster will have their TCP connections directed to the same cluster host. [Ref. 15]

2. WLBS Shortcomings

If one server fails, the WLBS detects the server failures within 10 seconds, but it doesn't switch a failed servers active connections to other servers. [Ref. 28] Therefore, active connections to a failed server are lost when the server goes off line; all other connections are not unaffected. Another drawback of the WLBS is its lack of data replication mechanism. Since the WLBS works at the network driver level and can't replicate data, it is only suitable for stateless services such as WEB server farm environment.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. LOCAL AREA FAULT TOLERANCE FOR SAAM SERVER

Fault tolerance for the SAAM server will be implemented in two phases: locally and remotely. The first phase, local area fault tolerance for the SAAM server, is focuses mainly on tolerating component failures of one server such as processor failure, disk failure and network interface card failure. The second phase, remote area fault tolerance (disaster recovery) for the SAAM server, backup servers are used to tolerate environmental faults such as fire, earthquake, and flood that cause unrecoverable server failures. The function of the second phase is to tolerate the failure of the local area fault tolerance implementation of the SAAM server. Disaster recovery for the SAAM server will be discussed in the next chapter.

Local area fault tolerance for the SAAM server will be implemented by one of the existing third party products. As the Windows NT operating system is more commonly used in the mission critical applications, many commercial companies are focusing on the enhancement of fault tolerance features for this operating system. Nowadays there are dozens of such products. In order to select a product that best meets the SAAM server fault tolerance requirements, we have examined the five most promising products. The findings are presented in this chapter.

A. PRODUCTS OVERVIEW

According to recent studies on products providing fault tolerance for Windows NT, the following five products are the most promising:

- ARCserve Replication 4.0 for Windows NT

- C0-Standby*Server* 4.2 for Windows NT
- Double-Take 3.0
- Endurance 4000
- Octopus 3.2

1. ARCserve Replication 4.0 for Windows NT

ARCserve Replication is a software product developed by Computer Associates (www.cai.com) to provide server resilience for the Microsoft Windows NT operating system. ARCserve Replication allows servers to be loosely coupled using existing network connections and requires no special-purpose hardware. [Ref. 25] ARCserve Replication has the following three components:

1. *Server component.* This component installs the ARCserve Replication Server service. This enables the computer to act as a primary or secondary server. The Server component must be installed on each computer that needs to be protected and on each server that will provide protection.
2. *Manager component.* This installs ARCserve Replication Manager. This is the user interface for the ARCserve Replication. It enables the user to set up server protection, monitor the progress of replication tasks, request a manual failover, and reinstate a failed server. These tasks can be performed for all ARCserve Replication servers from any computer running the Manager component. At least one Manager component must be installed.
3. *Alert component.* This installs the Alert notification system, and warns user whenever replication events (such as failovers) occur.

In order to provide the secondary server with up-to-date files, a process called *synchronization process* has to be performed. During the synchronization process, those parts of the primary file system that constitute the workload are accurately mirrored on the secondary server. This initial synchronization process typically requires large amounts of data to be transferred, which can be time-consuming [Ref. 25]. However, ARCserve Replication lets the primary and secondary servers continue to operate while synchronization takes place. Files that are open and in use can also be replicated.

When synchronization is complete, the secondary contains an up-to-date set of replicated files. When files are added or removed from a backed-up directory on the primary server, the changes are automatically replicated in nearly real-time on the secondary server. If a file is altered slightly, only the changes to the file are sent over the network to conserve bandwidth.

While changes are transmitted automatically between servers, the information is only committed to disk when a transaction is completed. This prevents database corruption should failover occur in midtransaction. The replication process takes place in the background and does not require client systems to close files. All data remains live and available during synchronization. Both FAT and NTFS are supported and the file systems on the two servers do not need to be the same replication. ARCserve Replication runs on existing hardware, provided that there is sufficient disk space for the replicated data.

To protect files on a server, a *replication task* must be set. The replication task defines the data to be protected, the secondary server to hold the replicated data, and the conditions that trigger a failover. The replication task also defines an alternate

identification for the primary server because the secondary server adopts the primary server identity while it is standing in for the primary. During this set up process, the user can also specify the level of protection desired (data replication with failover or data protection only without failover) and the speed of the underlying network (very fast, fast, slow, or very slow).

When the user sets up a replication task, certain conditions apply for clients if the primary and secondary servers are not in the same domain. Specifically following a failover, the clients can only access replicated data on the stand-in server if their accounts for the primary server also exist on the stand-in server. If the primary and the secondary servers are in the same domain, then user accounts automatically exist on both servers. [Ref. 24]

During normal operation, ARCserve Replication continuously monitors the state of the primary server via TCP/IP or IPX/SPX heartbeat messages over the network, looking for conditions that can cause it to initiate a failover. These conditions are as follows:

- Permanent loss of contact with the primary server
- Critically low disk space on the primary server
- On command from the system administrator

Permanent loss of contact with the primary server can have a variety of causes, including hardware and software crashes, power outage, and network malfunction. The secondary server monitors a regular "heartbeat" sent out by the primary. If a number of consecutive heartbeats are missed, ARCserve Replication assumes loss of contact (the precise period can be configured as it depends, among other things, on the network

speed). Optionally, it can then obtain more detail by using an independent serial connection between the primary and secondary servers, and by "pinging" other preselected network nodes. In this way, ARCserve Replication obtains sufficient information to determine whether loss of the heartbeat (as detected by the secondary) is due to genuine failure of the primary. This information, in conjunction with a policy set by the system administrator, helps ARCserve Replication to assess the severity of the failure, and whether or not to fail over. [Ref. 25]

The system administrator can configure the drives to be monitored, and the level of free space that is to be regarded as critical. ARCserve Replication can also monitor free disk space levels on the secondary server, and raise an alert and/or suspend the replication task if critical levels are reached. [Ref. 25]

The feature of failover on command from the system administrator is useful if planned maintenance or upgrade is required. The primary server can be taken out of service with little or no disruption. [Ref. 25]

ARCserve Replication offers a set of script files that execute automatically at the stages during failover. These script files can be used on both the primary server and the secondary server, before and after failover

During failover, the secondary server inherits the IP address and the NetBIOS name originally owned by the primary. In addition to this inheritance, the primary is given a temporary new address and a name to avoid conflicts in the network. The net effect of the failover is that in most case users and applications continue functioning and without interruption or requiring a new login. Some older applications that are not

designed for resilience may require the user to retry a file operation before continuing. [Ref. 25]

When the failed primary server has been repaired, the user runs the reinstate wizard. This wizard is not triggered automatically; the user can run it manually or schedule it to start at a specific time. The main steps during the reinstating process are as follows:

1. Optionally issue a warning message to logged-in users.
2. Synchronize the primary file system with the secondary so that changes made while the secondary was standing in are not lost.
3. Execute pre-reinstatement scripts on both the primary and the secondary server. These scripts might typically be used for closing down services, modems, etc.
4. Restore file shares and IP addresses to the primary server.
5. Execute post-reinstatement scripts on both primary and secondary servers, possibly to start up the services needed for users and applications.
6. Optionally restart the replication task, for continued protection.

2. Co-Standby*Server* 4.2 for Widows NT

Co-Standby*Server* for Windows NT was originally developed by Vinca Corporation, which was acquired by Legato Systems (www.legato.com) on August 2, 1999. Co-Standby*Server* is a clustering solution for Windows NT servers. Using Co-Standby*Server*, one can couple two Windows NT servers together to form a cluster. As

shown in Figure 4.1, in a typical configuration two servers are connected with a separate, dedicated network segment.

Active-active and *active-passive* configurations are two possible configuration types of the Co-Standby*Server*. In active-active configuration, each server is active on the network performing file and print functions and/or acts as an application server. Conversely, in active-passive environment, the second server does not perform any service on the network until the primary server's failure.

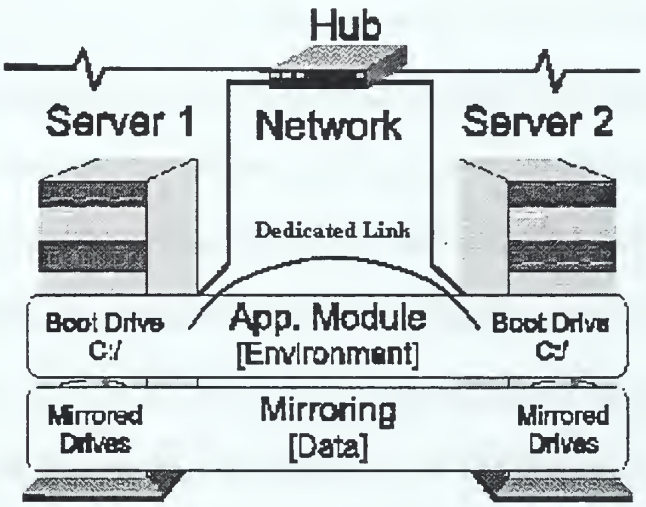


Figure 4.1 Typical Co-Standby*Server* Configuration.

During normal operation, a continuous bi-directional mirroring process sends data across the dedicated network segment ensuring that each server is kept up-to-date with data sets from both servers. Should either server fail, Co-Standby*Server* transfers critical functions from the failed server to the surviving server. This includes IP addresses, shares, print queues, server names and applications that were running on the failed server. Data that was mirrored from the failed server is now made available to the network through the surviving server. At the conclusion of the failover process all critical

network functions are now active on the surviving server and users can continue to access those functions with little or no interruption.

In Co-StandbyServer, only two servers can exist within a cluster. The two servers do not need to be identical. However, the two servers must have the same role in the network and must be members of the same domain.

During the installation process, Co-Standby*Server*'s setup installs five services and three device drivers.

Services:

1. *Disk Service*: Exports disk devices, which are redirected to the other server when a failover occurs.
2. *Event Manager*: Receives all errors and alerts for all Co-Standby*Server* services, devices, and drivers. These errors and alerts are logged in the applications Event Log and a text log.
3. *Vinca Service*: Monitors communication between the servers and controls the failover and failback process.
4. *Transport Service*: Provides communication service for the dedicated link between mirroring engines.
5. *Watch Service*: Watches changes in the Registry for clustered applications in order to replicate those changes to the other server.

Device Drivers:

1. *VincaFT*: Mirroring driver.
2. *VNCDisk*: Imports the disk devices that have been exported by the Co-StandbyServer Disk Service on the other server.

3. *VNCHint*: Provides transport services, which both the VincaFT and VNCDisk use to communicate between servers.

Installation of the Co-Standby*Server* components requires installing Windows NT on a separate drive, or logical drive within an array. Additionally, each server must have three physical drives (or three logical drives configured in a disk array) for an active-active configuration, or two drives per server for an active-passive configuration (configuration types will be defined later.)

Each block of data residing on the clustered volume is mirrored to the other disk device located on the other server forming a *mirrored set.* A mirrored set consists of two different partitions (on different disk devices and on different servers) logically combined to look like a single volume. If the I/O card or disk drives on one server fails, nothing happens to data access because there is still an active I/O card or disk device inside the mirrored set. This is the same benefit received if one is mirroring two drives internally on a server and a drive fails; users can still access data from the remaining drive in the mirrored set. If one of the disk devices in the mirrored set becomes unavailable for any reason, a delta file is generated that marks the blocks of data that have been changed since the drives were unavailable. The process used by the delta file is capable of holding as much as 2GB of changes in one 64K block. Therefore, it is unnecessary to allocate additional drive space for buffering changes when mirroring is prevented. [Ref. 21]

For a cluster server to take over the functionality of a server with mission-critical applications, it must have these application files and their support modules along with the same registry keys that make up the application. This can be achieved by application support scripts or can be done manually. To manually stage a registry, the user must

install the application to the other machine within the cluster using the same directory paths that were used when the application was installed on the original server. Command files can then be created that start the installed application on the other server when failover is processed. [Ref. 22]

Each server within the cluster maintains its own resources. If any of these resources are clustered using Co-Standby*Server*, they are associated with an alias NetBios computer name known as a *failover group*. Failover group is simply a server alias name container that can move between physical server hosts as needed. The failover group can be activated on either server within the cluster. Activating a failover group on a server allows users to see the alias computer name on the network that are pointing to the server hosting the clustered resources. [Ref. 21]

Co-Standby*Server* provides manual and automatic failover capabilities. Manual failover can be employed for scheduled maintenance or for load balancing purposes. In automatic failover, Co-Standby*Server* sends heartbeat checks across both the client network and the dedicated link between the servers (when used). Only a failure of both links causes a failure condition.

When a failure is detected, Co-Standby*Server* checks the properties of the Failover Groups. If there are any Failover Groups currently active on the failed server and they are configured to automatically failover, Co-Standby*Server* takes action on the surviving server to prepare it to receive the resources of the failed server. The resources of the failed server are then activated on the surviving server and the failover process is complete. Automatic failover occurs only in the event of a failure of one of the cluster servers. By definition, after an automatic failover, the cluster is no longer in a protected

state and the surviving server resources are at risk of another server failure. The automatic failover is in contrast to manually moving a failover group, which does not alter the availability state of the cluster. [Ref. 22]

When a server failure causes an automatic failover of a Failover Group, the cluster is considered to be in a failed over state because there is only one host server. This condition should be repaired as soon as possible in order to return the system to its original high-availability state. The necessary steps for recovery depend on what caused the server to fail. If a server hosting a failover group fails and the automatic failover option is armed, the failover group will be activated on the surviving server. If the automatic failover option is not armed, the failover group can be moved manually to the surviving server to activate the resources and make them available to users. In most cases when the server is repaired and returned to the cluster, it automatically resynchronizes the cluster volumes and is available for hosting the failover group. [Ref. 22]

3. Double-Take 3.0

NSI Software's (www.nsisw.com) Double-Take is a data replication and failover software product. The process begins by identifying the mission-critical data to be protected. The machine that holds the original copy of this data is known as the *source machine*. The selected data, known as the *replication set*, is then copied to another computer, known as the *target machine*. The target machine, on a local network or at a remote site, stores the copy of the critical data from the source machine.

After the target has a copy of the source's data, Double-Take monitors any changes to the data contained in the replication set and sends the changes to the target

machine through a process known as *replication*. Double-Take replicates only the changes rather than copying an entire file. [Ref. 18] The replication process does not require a dedicated link between the source and the target machine.

The failover module resides on the target server, and continually monitors the source servers. In the event of a server failure, the target server can assume the names and IP addresses of the failed servers (in addition to its original name), and invokes scripts to restart applications.

Double-Take components can be classified in two groups: server components and client components:

Server Components:

1. *Double-Take Service:* This service controls the core functionality of Double-Take including mirroring and replicating as well as failover functionality for the source machine.
2. *Server Monitor Service:* This service controls the failover monitoring functionality on the target machine.
3. *Logger Service:* This service controls the Double-Take logging utility. This utility logs alerts (notifications, warnings, and errors) that occur during Double-Take processing.

Client Components:

1. *Management Console*: This client is a graphical user interface where you can work with all aspects of Double-Take including failover configuration.
2. *Text Client*: This client is a full-screen, text-based client, which uses the Double-Take Command Language.

3. *Command Line Client:* This client is a line-by-line, text-based client that uses the Double-Take Command Language.
4. *Failover Control Center*: This client is a graphical user interface, which can be used to configure and monitor all aspects of Double-Take failover.

During the mirroring process, the Double-Take transmits the data contained in a replication set from the source to the target machine so that an identical copy of data exists on the target machine. All file attributes and permissions are also mirrored to the target machine. Mirroring must occur initially to generate a baseline copy from the source to the target. After mirroring has occurred, replication maintains an identical copy of the data on the target. Figure 4.2 shows the different steps that are completed when a mirror is performed. Mirroring process includes following steps:

1. Mirroring is initiated by the user, either manually through the one of the clients or automatically when the connection is created.
2. Double-Take determines which data needs to be sent to the target depending on the mirroring criteria that was specified through the client. If it is a full mirror, all of the files are immediately sent to the target. If it is a file differences mirror, the files contained in the replication set on the source are compared against the identical copy of the replication set on the target to determine which files need to be mirrored.
3. Double-Take transmits the mirror data to the target machine.
4. As each packet of mirror data is received on the target, the target returns an acknowledgment to the source confirming that the mirrored data has been received.

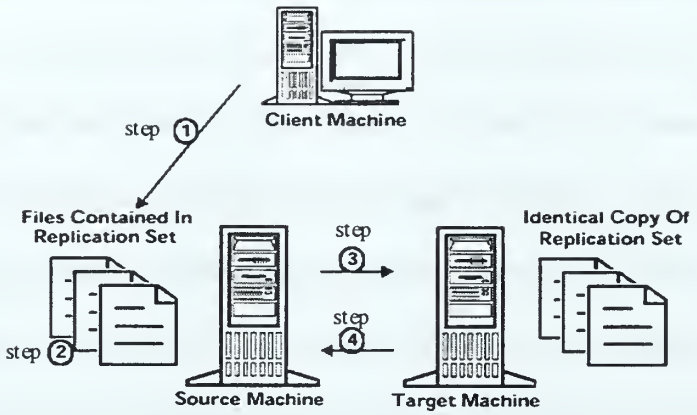


Figure 4.2 Double-Take Mirroring. [From Ref. 18]

Double-Take's replication process operates at the file system level and is able to track file changes independently from the file's related application. Once the source and target have been connected, Double-Take begins tracking file system changes that affect the data included in a replication set. During replication, Double-Take immediately records these file changes and groups them in packets. The packets are placed on a queue corresponding to each connection. Double-Take accumulates packets on the appropriate queue until the transmission of the packet to the target has been successful. When the target receives the packet, it responds with an acknowledgment and the source removes the acknowledged packet from the queue. Figure 4.3 shows the components that are involved in the replication process. Replication process includes following steps:

1. The operating system handles all file requests when an application creates, modifies, or deletes data on the source machine.

2. The file requests are intercepted by the Double-Take driver, *DblHook.sys*, on a Windows NT source machine or by the Double-Take File System (DTFS) on a UNIX source machine.
3. DblHook.sys or DTFS forwards all file requests to the file system and to Double-Take. The file system writes the operation to disk on the source machine and Double-Take converts the file requests into replication packets.
4. The Double-Take source sends the replication packets to the Double-Take target where they are applied to the target copy of the data.

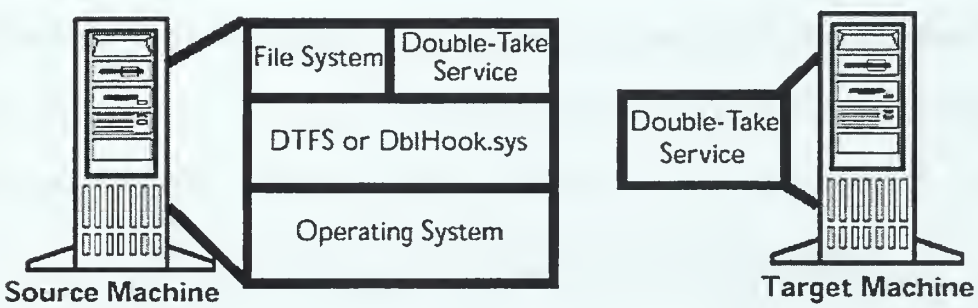


Figure 4.3 Double-Take Replication Components. [From Ref. 18]

Double-Take monitors the status of machines by tracking network requests and responses exchanged between monitored source machines and the target machine. The target sends a monitor request to each monitored IP address at a user-defined interval. A monitor reply is sent from the source back to the target. When the user-defined number of missed packets is met, the address is considered "failed." At this time, the failover process occurs or manual intervention is requested. In the event of a failover on a Windows NT machine, the target assumes the identity of the failed source including machine name, IP address, and subnet mask. Failover also send updates to routers and

other machines to update the IP to MAC address mapping. Network packets and applications destined for the failed IP address are routed to the target machine.

Depending on the type of client workstations, the timeout settings, and the applications in use, the clients may notice only a slight pause while the failover process occurs. If the failover timeout is set for a duration such as several minutes, clients may see an Abort or Retry message if they try to communicate with the source before the timeout has expired and the failover process has completed. For most clients and network aware applications, reconnection is automatic. By incorporating user-defined failover scripts into the process, network administrators can automate many network and application events on the target machine, such as starting applications or system services or sending network messages to administrators. [Ref. 18]

Double-Take's failover capability can be used on the following types of networks:

- Local Area Network (LAN)
- Wide Area Network (WAN) with Virtual LAN (VLAN)
- WAN with WINS or DNS reconfiguration
- WAN with source servers using secondary IP subnets.

On a LAN, Double-Take can failover servers without any additional network addressing concerns. During failover, the target server assumes the IP address and machine name of the failed source server while maintaining its own identity. After the target server assumes the identity of the source server, it sends out an Address Resolution Protocol (ARP) reply broadcast so that all machines on the LAN will send packets to the target server. This reconfiguration can be completely transparent to the clients. [Ref. 19]

Failover can also happen on WAN with VLAN. A VLAN is a group of devices that logically appear as local to each other, but are separated physically. The routers and/or switches in a WAN hide the true location of devices in a VLAN from each other. The routers and switches handle ARP requests to ensure all devices on a VLAN can see each other as local devices, regardless of the actual network distance between the devices. In a VLAN, the routers and switches allow Double-Take to failover across a WAN. It allows the source and target servers to be on the same logical IP subnet even thought they are separate. The Figure 4.4 shows IP addressing in a normal VLAN environment and the Figure 4.5 shows what happens during failover across VLAN. The CLIENT, will still see the server SOURCE as local even it is on a physically different network. [Ref. 19]

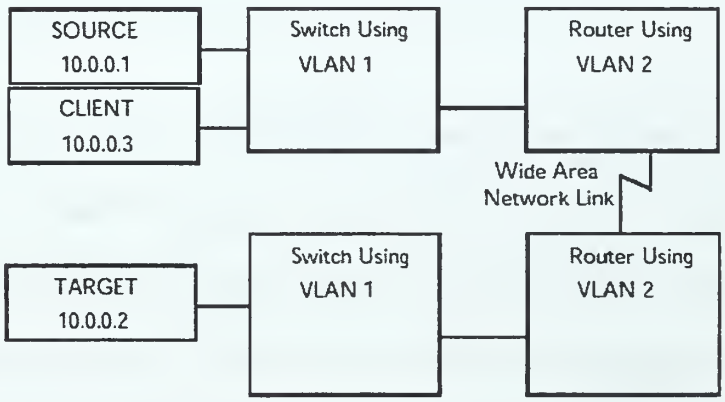


Figure 4.4 Normal VLAN Operations. [From Ref. 19]

In a typical WAN environment, an IP address that is valid on one network segment will not be valid on another segment. In this case Double-Take needs a way to change the IP address associated with a server name during the failover process. This

allows clients to send packets to the same server name, but to a different IP address. Changing the IP address associated with a server name can be accomplished with WINS or DNS scripting. [Ref.19]

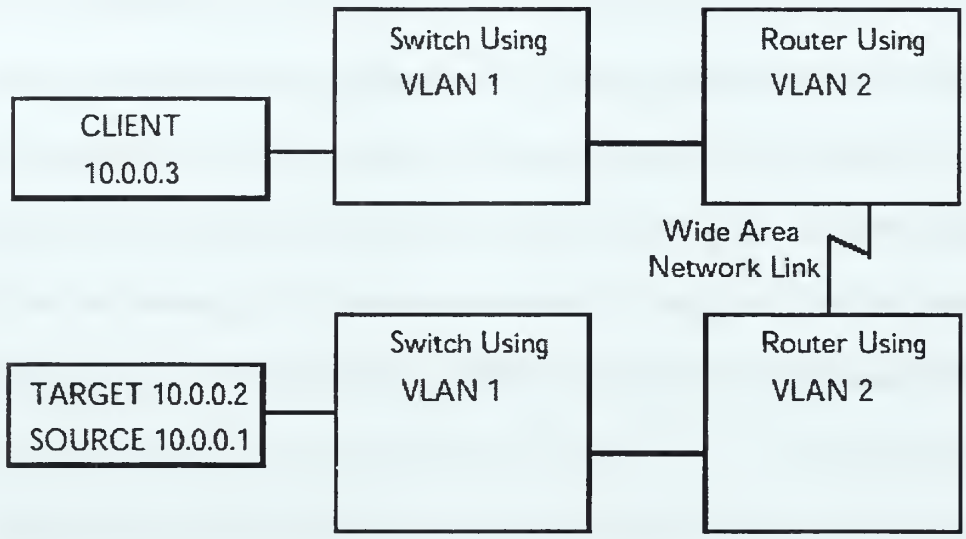


Figure 4.5. Failover on a VLAN. [From Ref. 19]

Another method of failover is to failover on WAN with source servers using secondary IP subnets. Using this method, each source server talks to a router via a unique secondary address on a router port. When failover occurs, the secondary address is moved from the router port associated with the source server to the router port associated with the target server. The routing protocols (i.e. RIP, OSFP, IGRP, EIGRP) in use will update all routers and let them know that the sub-net is now in a different location. Figure 4.6 shows a sample configuration before failover and Figure 4.7 shows the same configuration after failover (the sub-net mask used in the entire configuration is 255.255.255.0). [Ref.19]

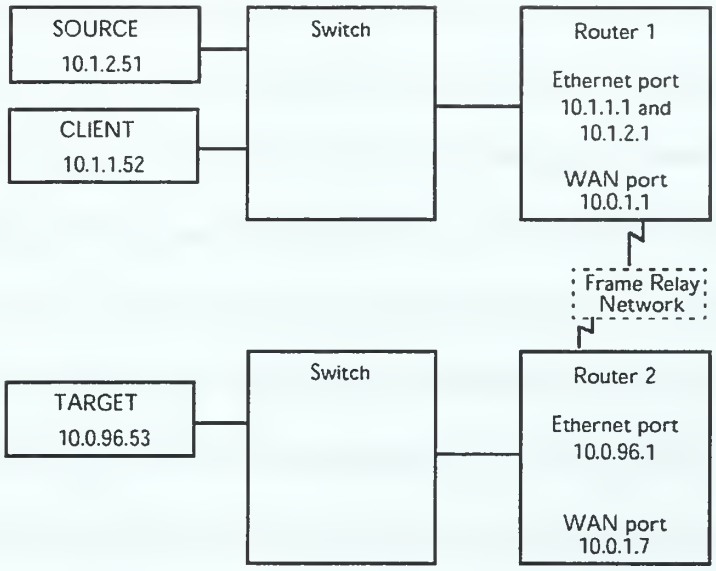


Figure 4.6. WAN Source Server Using Secondary IP (before failover.) [From Ref. 19]

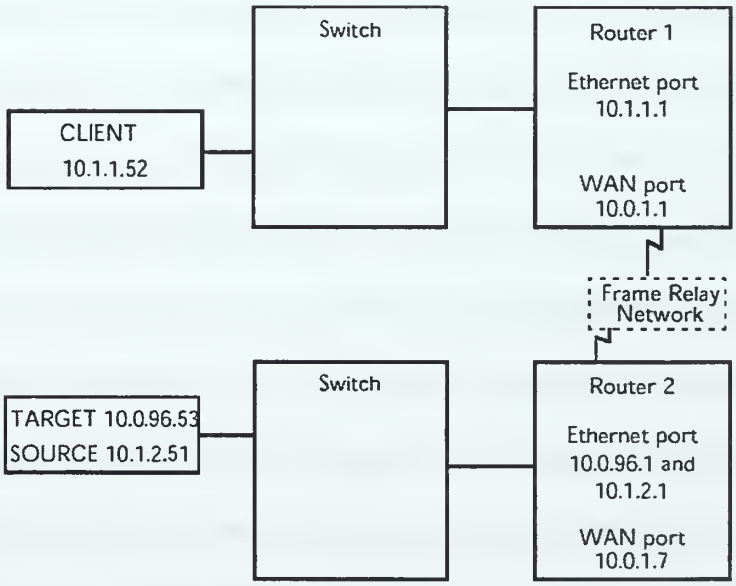


Figure 4.7. WAN Source Server Using Secondary IP (after failover.) [From Ref. 19]

After the problems of the failed server are corrected, the network administrator manually initiates a process called *failback*. The failback refers to the reinstating the repaired source server in the network. In order to avoid conflicts, the source machine should not be reattached to the network until Double-Take has completely removed the source's identity from the target. Depending on the type of machine and data that Double-Take is protecting, failback may need to be scheduled for an inactive period. If failover is being used in conjunction with Double-Take replication or if a drive on the source was replaced, the data on the source may not be current. It may be necessary to restore the most recent data from the target machine to the proper location on the source before initiating the failback process and bringing the source back online.

Users may notice an interruption at their workstations during failback. This delay will occur between the completion of the failback process and the time needed to bring the source machine back online. Like failover, network administrators can incorporate user-defined failback scripts into the process to automate many network and application events on the target machine, such as starting applications or system services/daemons or sending network messages to administrators. [Ref. 18]

Double-Take can be configured in various forms. As shown in Figure 4.8, sample configurations for Double-Take are as follows:

- *One-to-One, Active/Standby*: One target machine, having no production activity, is dedicated to support one source machine. The source is the only machine actively replicating data. This configuration is appropriate for offsite disaster recovery, failover, and critical data backup.

- *One-to-One, Active/Active*: Each machine acts as both a source and target actively replicating data to each other. This configuration is appropriate for failover and critical data backup and is more cost-effective than the Active/Standby configuration because there is no need to buy a dedicated target machine for each source.
- *Many-to-One*: Many source machines are protected by one target machine. This configuration is appropriate for offsite disaster recovery. Many-to-one configuration is also an excellent choice for providing centralized tape backup because it spreads the cost of one target machine among many source machines.
- *One-to-Many*: One source machine sends data to multiple target machines. The target machines may or may not be accessible by one another. This configuration provides offsite disaster recovery, redundant backups, and data distribution. For instance, this configuration can replicate all data to a local target machine and separately replicate a subset of the mission-critical data to an offsite disaster recovery machine.
- *Chained*: One or more source machines send replicated data to a target machine that in turn acts as a source machine and sends selected data to a final target machine which is often offsite. This is a convenient approach for integrating local high availability with offsite disaster recovery. This configuration moves the processing burden of WAN communications from the source machines to the target machines.

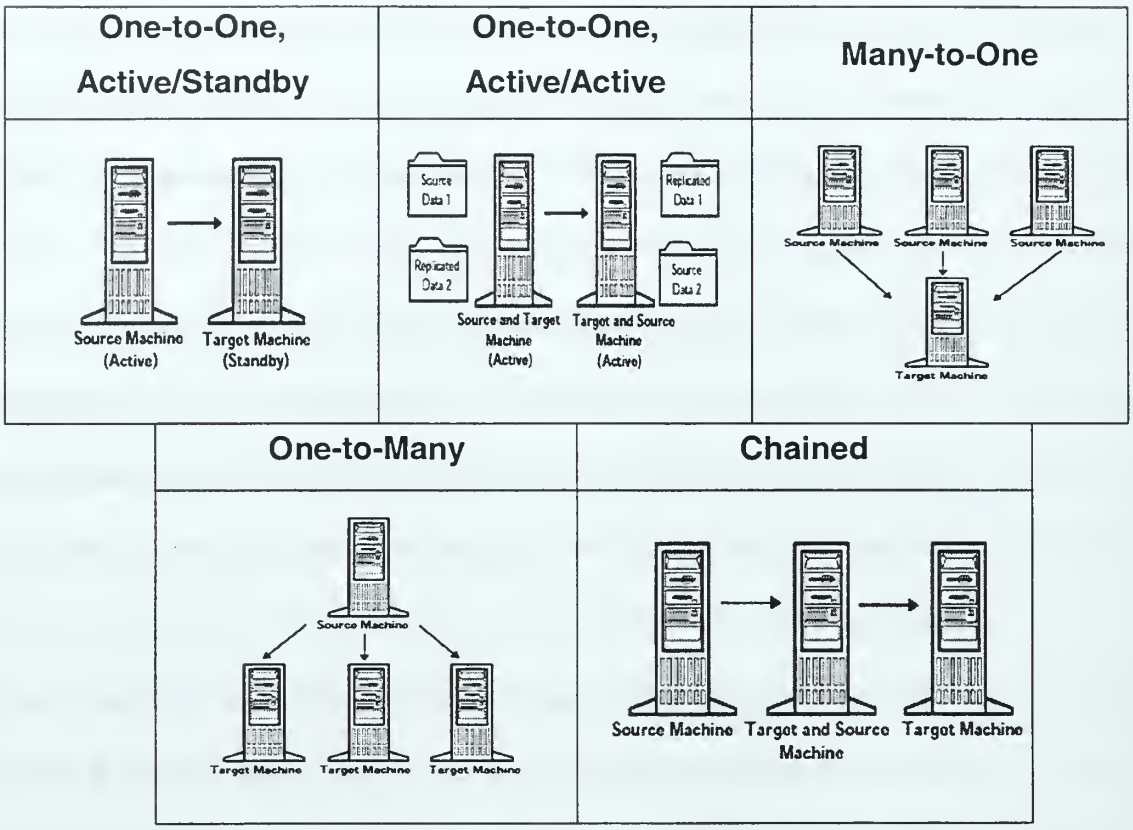


Figure 4.8. Double-Take Configuration Options.

4. Endurance 4000

The Endurance 4000 is developed by Marathon Technologies Corporation (www.marathontechnologies.com). It aims at providing 99.999% availability for the Windows NT server. Instead of the failover-based approach, Endurance 4000 uses hardware redundancy and duplicates execution and processing on multiple systems at the same time to make system failure transparent to application users. In Endurance 4000 hardware redundancy is achieved by combining four PCs into a one fault tolerant server as shown in Figure 4.9. The PCs are grouped into two *tuples* as shown in Figure 4.10.

Each tuple consists of a Compute Element (CE) and an Input/Output Processor (IOP) connected together. In each tuple one PC does the computation and the other one process the I/O operations.

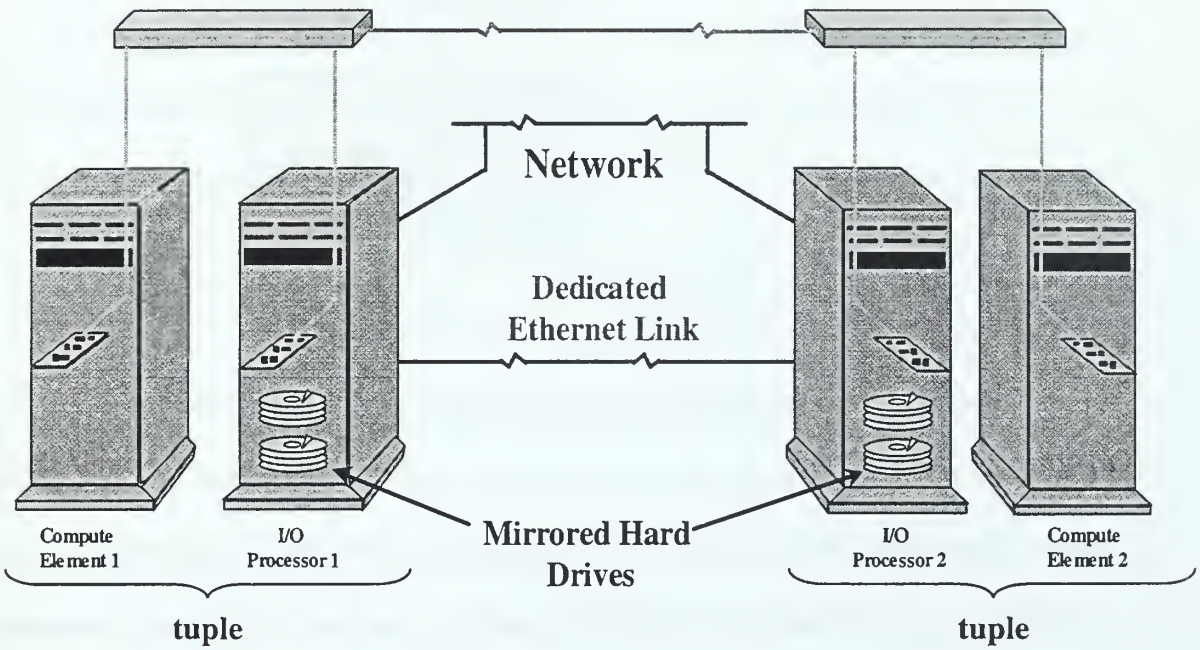


Figure 4.9. Configuration of the Endurance 4000.

Endurance 4000 consists of the following components: Four PCI-based Marathon Interface Cards (MICs) and interconnect cables, two SplitSite Data Links (SSDLs) and an Endurance software CD. There are four basic concepts underlying the Endurance technology that supplies and manages the necessary redundancy:

1. *Division of labor*: the user's computing tasks are logically and physically separated from I/O activity.
2. *System redundancy*: the system is configured redundantly, providing significant availability and data integrity.

3. *Marathon software*: the system performs tasks such as error checking, fault isolation, synchronization, and system management.
4. *SplitSite capability*: a portion of the Endurance system can be placed in a different geographic location, providing an instant "hot site" should one of the sites be rendered inoperable due to a disaster.

A Marathon Interface Card (MIC) is used to connect the systems together and to perform tasks needed to support fault tolerance and disaster tolerance. A network-like I/O redirector is used to redirect all I/O from the CEs to the IOPs. All operating system and application I/O calls are redirected to the IOPs for processing.

Both CE and IOP consist of standard Intel-based systems running the Microsoft Windows NT operating system. The CE is dedicated to running all application and operating system software, and has no I/O devices such as disks or LAN cards. The IOP is dedicated to performing all I/O device operations. Because the IOPs accommodate all of the system's mass storage and I/O, the CEs only need to contain a MIC, a high speed CPU, and the memory needed to run the operating system and user's applications. The CE's failure rate is much lower than a fully configured system because the components that fail most frequently, disk drives and network cards, are not present.

A tuple forms a single logical server, but actually consists of two servers, each running the Microsoft Windows NT operating system. With its MIC-based interconnect technology, the Endurance product provides the ability to use a single VGA display, keyboard, and mouse, which can be switched between the CE and IOP. [Ref. 23]

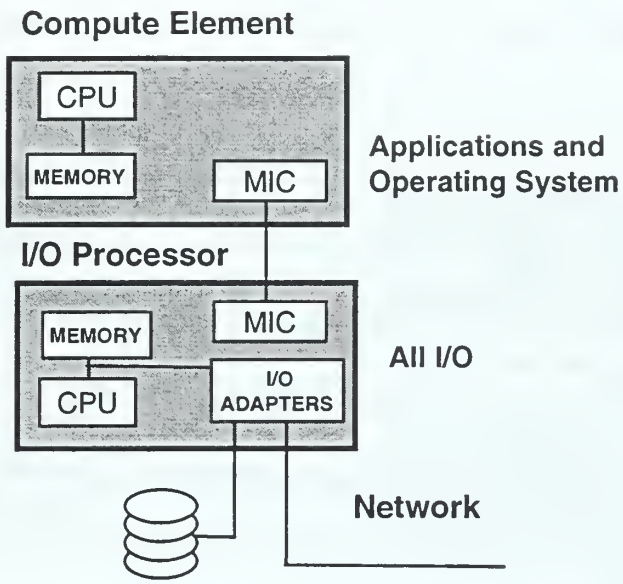


Figure 4.10. Marathon Tuple. [From Ref. 23]

In an Endurance 4000, system redundancy is provided both by the I/O and compute processing redundancy. I/O redundancy is implemented by using two I/O Processors, (as shown in Figure 4.11,). I/O to a disk used by the software running on the CEs occurs simultaneously on two disks, one on each IOP, ensuring that two copies of the data are always available. This configuration forms a RAID Level-1 type storage system without the need for a special RAID controller.

Similarly, there are two network interface cards, one on each IOP, to support application network I/O for applications running on the CE. This provides the redundancy required to guarantee network access to the applications running in the CE, in the presence of a failure of a network card. The 100BaseT interconnect between the two IOPs providing a path for communications between the IOPs to isolate and manage

failures. This path is also used to re-mirror a failed disk or system after repair. Re-mirroring occurs as a background task and is invisible to the user. [Ref. 23]

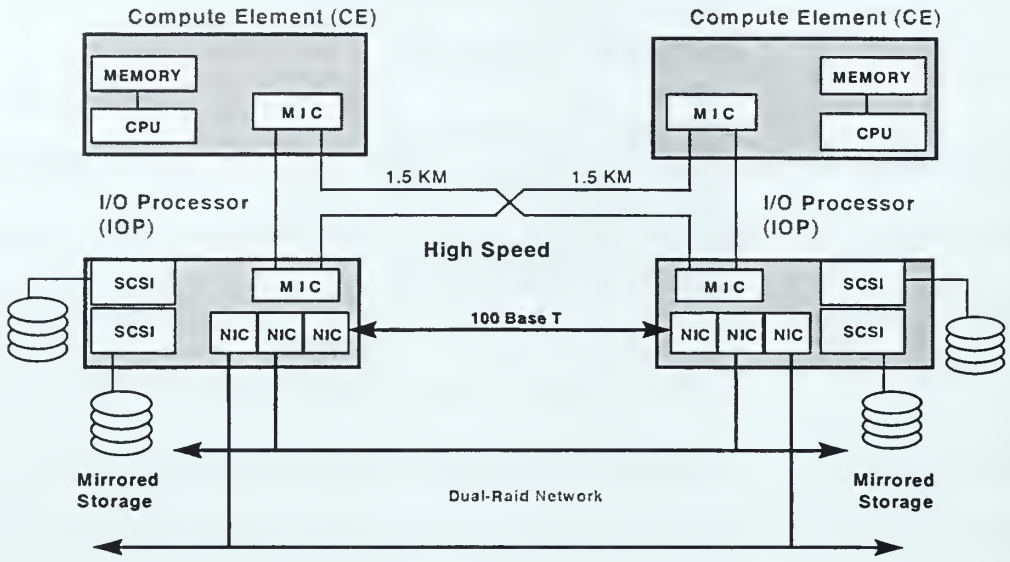


Figure 4.11. Endurance 4000 Array. [From Ref. 23]

The Compute Elements run in lockstep so that the failure of one CE is completely invisible to the end user as the remaining CE continues to compute through the failure. Also, subsystems can be repaired or upgraded while the system is online.

Marathon software is comprised of the following components: *I/O handlers*, *monitor*, *synchronization* and *Marathon System Manager*. The I/O handlers manage all the data movement between the hardware components of a Marathon system. They intercept all I/O requests in the CE and forward them to the IOPs, as well as receive the I/O requests from the CE and send them for processing to Windows NT drivers in the IOPs. All of these operations are done through the MIC, which also checks and compares data. [Ref. 23]

The Monitor runs in the IOPs. It manages the flow of data through the IOPs as well as coordinates the activities of the IOPs. It removes a subsystem when faults occur and, when a subsystem is repaired, the Monitor joins and restores the repaired subsystem into the Endurance configuration. [Ref. 23]

The fault handling software detects and isolates faults using the following two techniques. First, the software handles hard failures that can be isolated using timers and error detectors. Second, the software follows a rule-based system that uses past history and a set of rules to identify failures that are more difficult to isolate.

Marathon's synchronization software runs in the CEs allowing both CEs to function in lockstep. If one CE fails, the synchronization software removes the failed CE from the Endurance system and enables the other CE to maintain complete context and functionality of the operating system and all applications. After the failed CE is repaired, the CEs return to lockstep using software synchronization. [Ref. 23] Although any failed component can be repaired without interrupting the service, software running on the Compute Elements cannot be upgraded while the server is running. Scheduled down time is required to enable complete upgrades.

The Marathon System Manager (MSM) is a GUI-based management tool for the Endurance system. Using MSM, a user can manage, monitor, and configure the system remotely or from the local Endurance system (i.e., it can run on each CE or IOP and provide a view of the entire Endurance configuration as seen by each CE and IOP.) MSM allows visibility to the current status of an Endurance system and allows the system manager to monitor and correct failures. [Ref. 23]

Marathon's *SplitSite* technology allows Endurance tuples to be placed in different geographic locations, providing the user with an instant "hot site" should a disaster occur. As shown in Figure 4.11, the two system tuples can be separated up to 1.5 kilometers with Marathon's SplitSite link. [Ref. 23]

Marathon's architecture not only provides protection against hardware faults but also provides protection against transient software faults. It can also detect operating system failures and automatically restart the system. The I/O Processors (IOPs) run Marathon's I/O management and the fault handling software, leaving this portion of the system isolated from the loads placed on the Compute Element (CE) by the user's applications and operating system.

Further, since the IOPs handle all interrupts, the CE, (where the operating system and user applications are running,) is not subjected to the usual stream of interrupt asynchrony. Here, interrupts are managed through a structured process that eliminates a major source of asynchrony-induced software failures. Although the IOPs are subjected to these asynchronies, since there are two autonomous IOPs in a full fault tolerant system, an interrupt-induced software asynchrony will likely only affect one of the IOPs. [Ref. 23]

This isolation and the structured nature of the I/O software environment in both the CEs and the IOPs masks transient software faults, thus delivering a level of operating system software fault tolerance unavailable in any other system. Although masking transient software faults does provide more protection from software faults than has been previously available, the likelihood that an operating system software bug could cause an interruption of service is not eliminated. Since transient software faults account for only

75% of bugs found in typical commercial operating systems, there are still some bugs, which may cause the CE operating system to crash, delivering the now famous Windows NT "blue screen of death" to the user. In this event, the only option is to reboot the system.

Since the I/O Processors are independent processors that have full visibility of the activities in the CEs as well as control over them, they can be programmed to reboot the operating system running on the CE. At the user's option, the Marathon software can be configured to reboot the operating system automatically, making the system available to the users until the same software bug shows up again. It should be noted that the Marathon architecture also offers the IOPs the opportunity to observe the operation of the applications running on the CEs, and to restart a "hung" application. [Ref. 23]

5. Octopus 3.2

Octopus was originally developed by FullTime Software Corporation, which was acquired by Legato Systems (www.legato.com) on April 1, 1999. Octopus is a software-only solution for Windows NT that provides data mirroring and failover capability. Its aim is to increase both the availability and the reliability of Windows NT servers. Octopus operates over both Local Area Network (LAN) and Wide Area Network (WAN) connections and allows remote administration and installation. Users install Octopus on each Windows NT machine that will be used as a source and/or target. Octopus runs over any network interface supported by Windows NT and does not require a dedicated NIC. However, dedicated NICs can also be used to allow users the option of keeping Octopus-related traffic off their networks.

Octopus provides the following software components:

1. *Octopus Client*: This is the graphical user interface (GUI) of the product. It is used to configure Octopus, start Octopus services, and administer the Octopus. The Client provides capabilities for remote administration of Octopus.
2. *Octopus Service*: This component provides capabilities for switch-over and data mirroring. This component runs as a service in the Windows NT server.
3. *Octopus Device*: This component works with the Octopus Service to provide capabilities for data mirroring.
4. *Octopus SETUP and UNINSTALL programs*: These programs are provided for installing and removing Octopus on supported Windows NT platforms.
5. *SNMP Agent Extension DLL*: Extends the standard Windows SNMP service to allow Octopus to send its messages as Simple Network Management Protocol (SNMP) events, allowing Octopus to work with systems management software.
6. *Performance Monitor DLL*: Extends the Windows NT Performance Monitor to provide performance statistics on Octopus in the Windows NT Performance Monitor.

Octopus provides three functions that allow Octopus to mirror protect data including: (1) *Mirroring*, which captures changes in data at the source system, (2) *Forwarding*, which sends changes in data from the source system to the target system, and (3) *Updating*, which applies changes as stored in the receive log on the target system to the files on the target system.

Octopus updates target systems with changes in data as they occur, rather than re-sending all of the data at once. On each source machine, the user specifies which drives, directories and/or files they wish to replicate and the target system where the replicated files will reside. As shown in Figure 4.12, each time a change to a specified file is committed to disk on the source machine, Octopus mirrors it to the Octopus *send log*. Octopus then uses any Windows NT supported protocol to forward the change across the network to the Octopus *receive log* on the target machine. Finally, Octopus writes the change to the appropriate file on the target drive. [Ref. 20]

Octopus can mirror data in one-to-one, one-to-many, many-to-one or many-to-many configurations. As a result of this configuration options, it can be used as a data transport mechanism for data distribution and localization systems, in addition to data protection for applications. Conversely, applications that require collecting data at remote locations and continuously forwarding it to a centralized site can employ Octopus's many-to-one replication configuration. [Ref. 20]

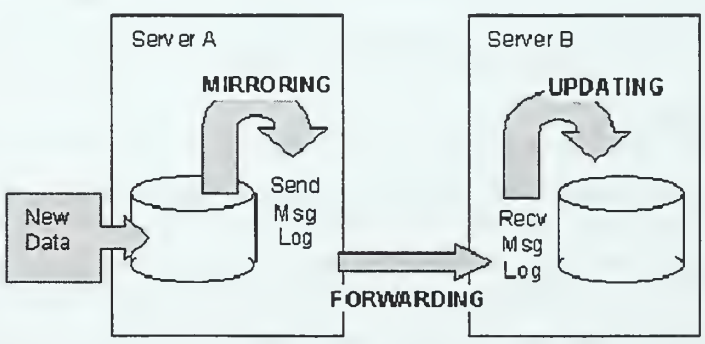


Figure 4.12. Data Protection Operation in Octopus. [From Ref. 20]

Octopus offers two kinds of switch-over (failover) capability, Automatic Switch-Over (ASO) and Super Automatic Switch-Over (Super ASO). The difference between

ASO and Super ASO is that Super ASO provides three additional capabilities [Ref. 20]: First, the target system can maintain its original identity while also assuming the identity of the failed source system, so that any clients using the target do not have their services interrupted. Second, the target system can simultaneously assume the identity of an unlimited number of source systems. Third, forwarding can continue after a Super ASO switch-over; with ASO, forwarding must cease since the original name of the target system "disappears" from the network.

On the source machine users configure a heartbeat between the source and target machines. The heartbeat identifies how often the source system sends "I'm alive," messages and how long the target system should wait after seeing the last "I'm alive," message before initiating the switch-over process. If the target system does not receive an "I'm alive," message from the source system within the specified time, it checks the Windows NT registry and service database for the source machine. If Windows NT can find the source machine on the network, the target machine resumes monitoring the "I'm alive," messages. If not the target machine initiates the switch-over sequence. In larger networks it may take longer to search the Windows NT registry and service database for the source machine.

During the switch-over process, the target machine assumes the machine name and, if specified by the user, the IP address(es) of the source machine. Users can identify services and/or applications to stop or start before or after the switch-over process has completed. After switch-over, the target can maintain its own identity as well as the identities of failed source machines. Users on the network can continue to work unaware that their server has failed and that the target machine has taken over. [Ref.20]

B. RECOMMENDATIONS

MSCS, WLBS and third-party products presented above all try to tolerate server failures and provide highly available services to clients within the Windows NT server environment. In selecting one of these products for the local area fault tolerance of the SAAM server, our main concern is to find out which one best meets the SAAM server fault tolerance requirements.

MSCS provides clustering solutions only to applications that are MSCS-aware. In other words, in order to benefit from MSCS, the application to be protected must be written using the specific API. Moreover, in an MSCS cluster, the two servers can typically be no further apart than allowed by the shared SCSI bus. The maximum distance is about 25 meters, mandating that the location of the standby server be in the same room. If there is a fire or a power outage, it is likely the whole cluster will be shut down.

Furthermore, Microsoft claims that MSCS recovers from a server failure in around one minute. However, the actual time needed for the recovery depends on the application type. Some users of MSCS complain that the failover process is too long for some applications, in some cases taking more than 30 minutes. [Ref. 29] Because of these drawbacks, MSCS does not satisfy the SAAM server fault tolerance requirements.

WLBS is designed for high availability and scalability of TCP/IP-based services such as Web servers, streaming media, Virtual Private Networking (VPN), and proxy—services generally considered to be "stateless." However, the SAAM server provides statefull services to routers. In other words, the service provided to routers by the SAAM server totally depends on the data residing on the PIB, which is updated frequently.

Therefore WBLS cannot be considered as a solution for the local area fault tolerance for the SAAM server.

Since the products bundled with the Windows NT Server Enterprise Edition could not meet the local area fault tolerance requirements of the SAAM server, a solution must be selected from the third party products. In order to compare and contrast these products, we classify them into two groups according to their approaches used for tolerating server failures. Among these third party products, ARCserve Replication, Co-StandbyServer, Double-Take and Octopus use the failover approach whereas, Endurance 4000 uses the hardware redundancy approach for tolerating the server failure.

Specifications of the third party products that use the failover approach are summarized in Table 4.1. Considering the failover times of these products, it is apparent that the offered failover time is between 30 and 45 seconds, which is too long for the SAAM server. Consequently none of these products are qualified as a solution for the local area fault tolerance for the SAAM server.

On the other hand, by using hardware redundancy Endurance 4000 can tolerate server failures in less than a second. Marathon's Endurance 4000 product allows industry standard Intel based PC systems to be configured as fault tolerant servers. The Endurance 4000 server runs the standard Microsoft Windows NT operating system and applications. This means absolutely no modifications, scripts, or APIs are required for SAAM Server applications.

Using SplitSite technology, the PC systems connected by Endurance 4000 can be placed at different locations up to 1.5 kilometers apart, while they operate and appear to

users as a single fault tolerant server. This provides continuity of service in the face of localized disasters that are confined to one building.

For these reasons, especially its ability to recover from server failures in milliseconds, we believe that Endurance 4000 best meets the criteria for the local area fault tolerance for the SAAM server. The main drawback of the Endurance 4000 is its price. Endurance 4000 costs $25,000, and the price does not include the four servers and multiple copies of Windows NT software. Although it is expensive, the price is justified when compared to the large cost of routers, switches, and networking software, and the amount of revenue at stake. Consequently, among all products discussed thus far, the Endurance 4000 is recommended as a solution for the local area fault tolerance for the SAAM server.

	ARCserve Replication 4.0	Co-Standby*Server* 4.2	Double-Take 3.0	Octopus 3.2
Price [1]	$2995	$4499	$3750	$2998
Built-in replication	Yes	Yes	Yes	Yes
Replication file updates only	Yes	No[2]	Yes	Yes
File/directory level selection	Yes	No	Yes	Yes
Replication before write-through to disk	Yes	No	Yes	Yes
Open file mirroring and replication	Yes	Yes	Yes	Yes
Limitation of bandwidth usage on the network	No	No	Yes	Yes
One-to-many replication	Yes	No	Yes	Yes
One-to-many failover	No	No	Yes	Yes
Allows dissimilar hardware and drive configurations	Yes	No	Yes	Yes
Allows custom Scripts or batch files	Yes	Yes	Yes	Yes
System requirements	•Intel x86 or better •6MB of disk space •32MB of RAM •One NIC per server	• Intel x86 or better • 30MB of disk space • 32MB of RAM • 3 physical hard disks (active/active) • 2 physical hard disk (active/passive) • One NIC per server (two is recommended)	• Intel x86 or better • 40MB of disk space • 16MB of RAM • One NIC per server	• Intel x86 or better • 15MB of disk space • Additional free disk space on target machines equal to size of replicated files plus 10% • 32MB of RAM • One or more NICs per server
Failover time[3]	45 seconds [Ref. 30]	30 seconds [Ref.29]	45 seconds [Ref. 30]	30 seconds [Ref. 31]
Suitability for SAAM (local area ft.)	No	No	No	No

Table 4.1. Specifications of Products, Implementing Failover Approach.

[1] Prices are for two server configuration and bases on [Ref. 30]
[2] Entire disk block must be transmitted
[3] Failover time is based on the SQL server failover

V. REMOTE AREA FAULT TOLERANCE FOR SAAM SERVER

This chapter focuses on the remote are fault tolerance for SAAM server and consists of three main sections. In the first section, first, the overview of the designed model is presented. After that, the details of the designed model are discussed according to the fault tolerance phases explained in Section II.D. In the second section, the integration of the model with the existing SAAM server source code is explained. Finally, in the third section, testing of the implementation is discussed. First, the testbed is explained, and then the test results are presented.

A. MODELING

Remote area fault tolerance of the SAAM server is provided primarily using the redundancy approach. Specifically, a redundant backup server is used (see Figure 5.1). Routers are required to send updates to both the primary and the backup servers. The backup server maintains its own PIB in parallel with the primary server. However, the backup SAAM server does not respond to router's requests until it detects a failure of the primary server.

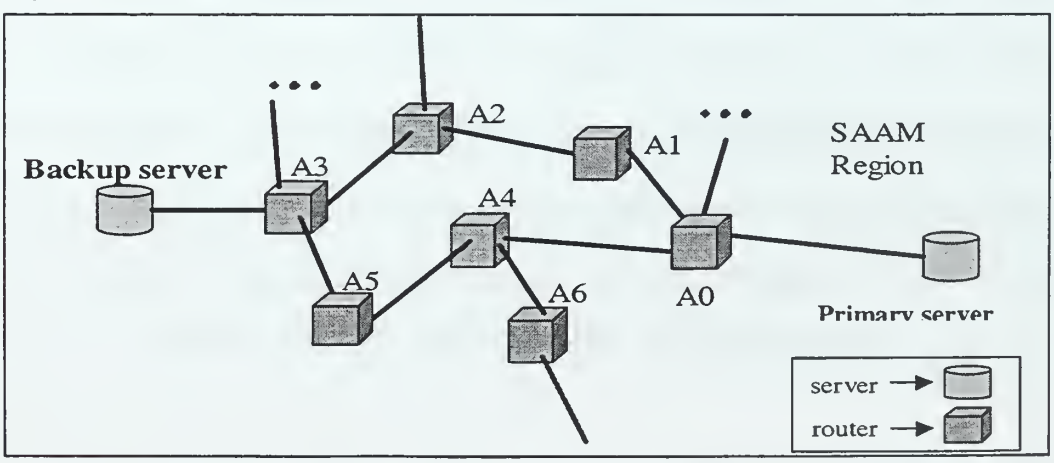


Figure 5.1. Server Positioning in the SAAM Region.

1. Server States

For the proposed primary-and-secondary approach, the states of the SAAM server and the state transitions are identified and illustrated in Figure 5.2. Both the primary and the backup servers begin their operation from the *initial state*. The initial state refers to the state of the server node* prior to the *server software agent* installation. The server software agent is a mobile Java class file sent by the system or network administrator, and installed on the server node. With the server software agent, the server node can be initialized either as a primary server or backup server.

Upon installation of the primary server software agent, a server node becomes the primary server and enters an *active running state*. In the active running state, the primary server provides services to routers such as flow routing table entry updates and flow responses.

Upon installation of the backup server software agent, a server node becomes the backup server and enters a *silent running state.* In the silent running state, the backup server maintains its PIB the same way as the primary server, but does not respond to routers' requests. Additionally, the backup server continuously monitors the health of the primary server. The mechanism for such monitoring is described in Section 2. When the primary server fails, it effectively enters a *failed state*. When the backup server detects a failure, it enters the active running state and takes over the functionalities of the primary server. While the primary server is running, if the backup server fails, then the backup server enters the failed state.

---

* Server node is a host on the network that will serve either as a primary or as a backup SAAM server.

The failed primary server stays in the failed state until it is corrected. After repair, the administrator may choose to reinstate the repaired server to the network either as a primary or as a backup. If the administrator wants to reinstate the repaired server as the backup, then the repaired server enters the silent running state. However, if the administrator wants to reinstate the repaired server as the primary, then the repaired server enters a *PIB synchronization state*.

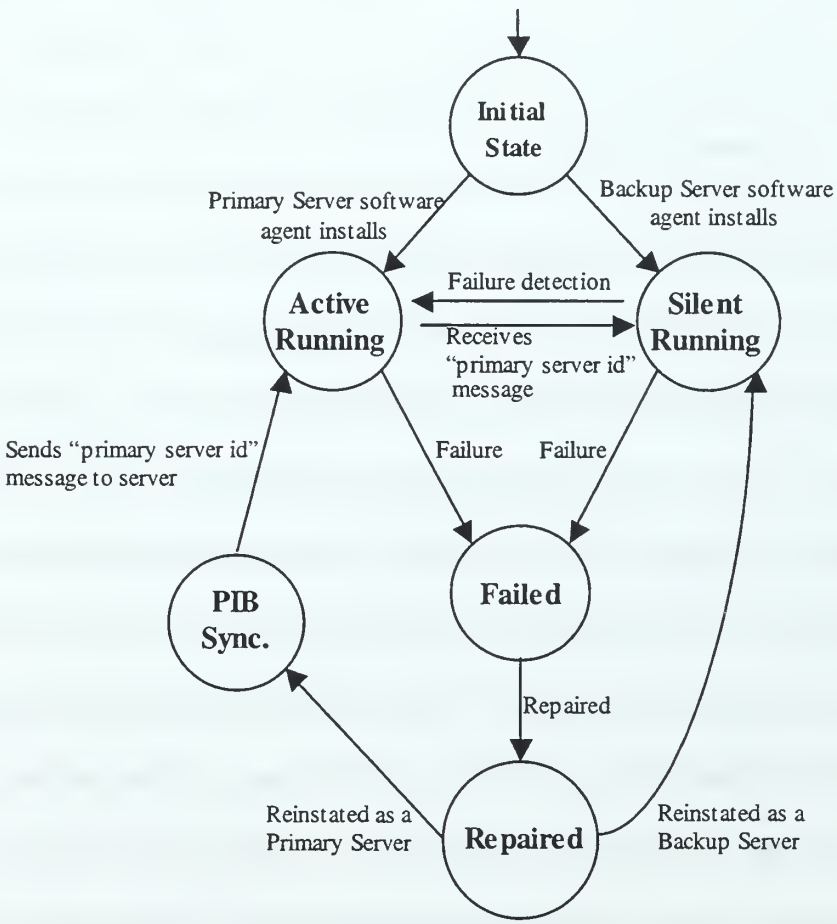


Figure 5.2. State Transition Diagram of a SAAM Server.

In the PIB synchronization state, the repaired server rebuilds its PIB from the link state advertisement (LSA) messages received from the routers. After the PIB synchronization is completed, the server sends a "primary server id" message to the currently active primary server. Then, the repaired server enters the active running state, while the backup server returns to the silent running state.

There are four major phases in our design for remote are fault tolerance of the SAAM server: *failure detection*, *damage confinement and assessment*, *error correction* and *fault treatment and continued service*.

2. Failure Detection

In most distributed systems, failure of a system component is detected by implementing a periodical message exchange mechanism among the system components. This type of message exchange mechanism is called *heartbeat protocol*. Specifically, a working component must periodically emit "beat" messages to show that it is operating properly. If the component fails to emit a beat message within the timeout period, then its failure is detected. Since heartbeat protocols check timing related constraint of a system, they can be considered as an example of timing checks, discussed in Chapter II.

Many different uses of heartbeat protocols are reported in the literature. For example, they are used in process termination in distributed programs (if a process an a program terminates or fails, then the remaining processes in the program terminate) [Ref. 26], network protocols [Ref. 27], reaching agreement on processor-group membership [Ref. 33], and mobile computing [Ref. 34]. In the model designed in this thesis, a heartbeat protocol is also used for the SAAM server failure detection.

When designing a heartbeat protocol, the protocol designer should strive to incorporate the following three ideal characteristics into his design [Ref. 26]:

1. The rate at which heartbeat messages are sent in the protocol should be small in order to reduce protocol overhead.
2. The detection delay (time difference between the detection time and the actual failure time) should be small, in order to improve protocol responsiveness.
3. The probability of false detection should be small, in order to increase protocol reliability.

In any heartbeat protocol implementation, it is impossible to incorporate all these three objectives at the same time, because they are somewhat contradictory. For example, to reduce both the rate of sending heartbeat messages and the detection delay, the protocol should allow only a small number of missed heartbeat messages. In this case, probability of false detection would increase. Therefore, every heartbeat protocol is a compromise between these contradictory objectives [Ref. 26].

Remote area fault tolerance for the SAAM server is mainly focused on tolerating environmental faults such as fire, earthquake, and flood, which cause unrecoverable server failures. Therefore, it is essential to locate the primary and the backup SAAM servers as much apart as possible in the SAAM region, as shown in Figure 5.1. In heartbeat protocol implementations, it is usually preferable to use a dedicated link between the two servers. By using such a dedicated link, the protocol overhead introduced to the network can be avoided. However, in SAAM, using a dedicated link between two servers is not practical, because of the long distance between the two

SAAM servers. Therefore, the heartbeat protocol to be implemented will use the existing network links for communications between the two servers.

In order to select the best heartbeat protocol for detecting the SAAM server failures, two different heartbeat protocols that cover most of design space, *constant heartbeat protocol* and *accelerated heartbeat protocol,* are prototyped and their performance results are compared. The following sections will discuss these two heartbeat protocols, their implementations, and their performance results.

a. Constant Heartbeat Protocol (V.0)

The constant heartbeat protocol is the first protocol that we prototyped and evaluated. Therefore, it was given a version number zero (V.0). Additionally, due to the constant rate of heartbeat messages generated by this protocol, it is called "*constant heartbeat protocol.*"

In the constant heartbeat protocol, the primary SAAM server periodically sends heartbeat messages to the backup SAAM server, indicating that it is in an operational state. On the other hand, the backup SAAM server listens only to these heartbeat messages coming from the primary server. If the backup SAAM server misses a predetermined number of consecutive messages, then it declares the failure of the primary SAAM server.

Let the time interval for sending heartbeat messages from the primary SAAM server to the backup SAAM server be t, and the maximum number of consecutive message misses allowed (allowed-miss) be n. The following equation relates t, n, and the detection delay denoted by d:

$$(t \cdot n) < d < (t \cdot (n+1)) \tag{5.1}$$

Figure 5.3 illustrates this relationship. In this case, n is set to two. According to the figure, after the first two heartbeat messages, the backup SAAM server did not receive any heartbeat messages. Since n is equal to two, after three consecutive misses, the backup SAAM server declares the failure of the primary SAAM server. The actual failure, if there is one, must have occurred at some time during the second interval. Therefore, the vertical arrows labeled d_{max} and d_{min} show the possible maximum and minimum detection delay times, respectively. In other words, detection delay d, should be less than $3t$ but greater than $2t$.

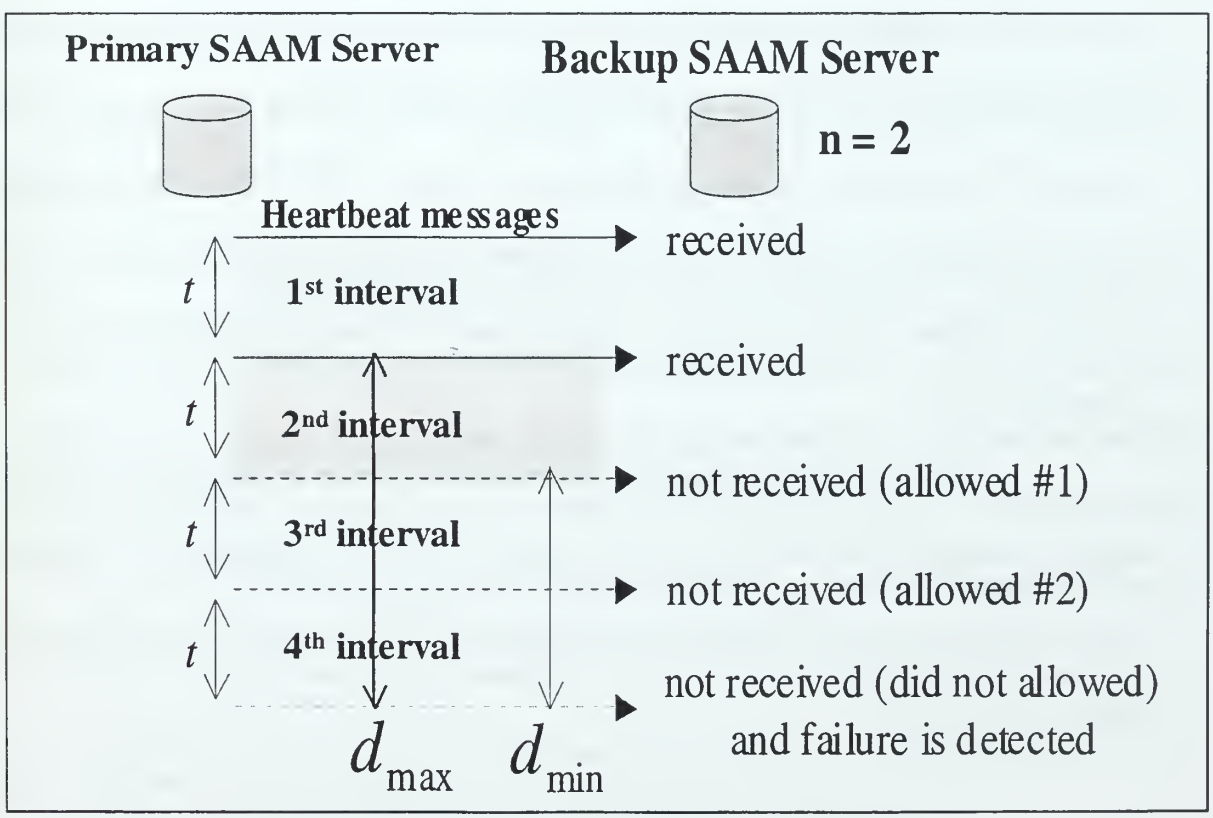


Figure 5.3. The Detection Delay in the Constant Heartbeat Protocol.

b. Accelerated Heartbeat Protocol (V.1)

The concept of accelerated heartbeat protocol was introduced by Godua and McGuire in 1998 [Ref. 26]. They use such protocols as a process termination mechanism in distributed programs to ensure that if a process in a program terminates or fails, then the remaining processes in the program also terminate. The same concept is adapted to the SAAM server as failure detection resulting in the development of the second version (V.1) of SAAM server heartbeat protocol, which we refer to as the *accelerated heartbeat protocol.*

In the accelerated heartbeat protocol, the communication between the primary SAAM server and the backup SAAM server is partitioned into successive time periods. At the end of each period, the backup SAAM server actively queries the health of the primary SAAM server by sending a message called *heartbeat query message* ("Are you alive?"). After that, the backup SAAM server waits to receive a message from the primary SAAM server called *heartbeat response message* ("Yes, I am alive"). The received heartbeat response message indicates that the primary SAAM server is in an operational state.

As long as the primary SAAM server stays in an operational state, the length of the period, denoted by t_{max}, is constant. However, the length of the next period can vary depending on the events that has occurred in the current period according to the following three rules:

1. The backup SAAM server sends a heartbeat query message to the primary SAAM server and receives a heartbeat response message

within the first half of the current period. In this case, the backup SAAM server makes the length of the next period t_{max} (irrespective of the length of the current period.)

2. The backup SAAM server sends a heartbeat query message to the primary SAAM server, but does not receive a heartbeat response message within the first half of the current period. In this case, the backup SAAM server immediately sends another heartbeat query message, and reduces the length of the next period by half.
3. The length of the next period ever becomes less than a specified value t_{min}, which is an upper bound on the round-trip network delay between the backup SAAM server and the primary SAAM server. In that case, the backup SAAM server declares the failure of the primary SAAM server.

If we assume that t_{min} is set to allow three consecutive heartbeat response misses, then the operation of the accelerated heartbeat protocol would be as shown in Figure 5.4. After the first heartbeat query message, the backup server has received the heartbeat response message. Thus, it is certain that the primary server was alive at time $(t_0 + (t_{min} / 2))$. Additionally, since the backup server did not receive a heartbeat response message for the heartbeat query message sent at t_1, the failure of the primary server must have happened before the second heartbeat query message. Consequently, the actual failure of the primary server must have happened at some time between $(t_0 + (t_{min} / 2))$ and t_1, which is the shaded area in Figure 5.4.

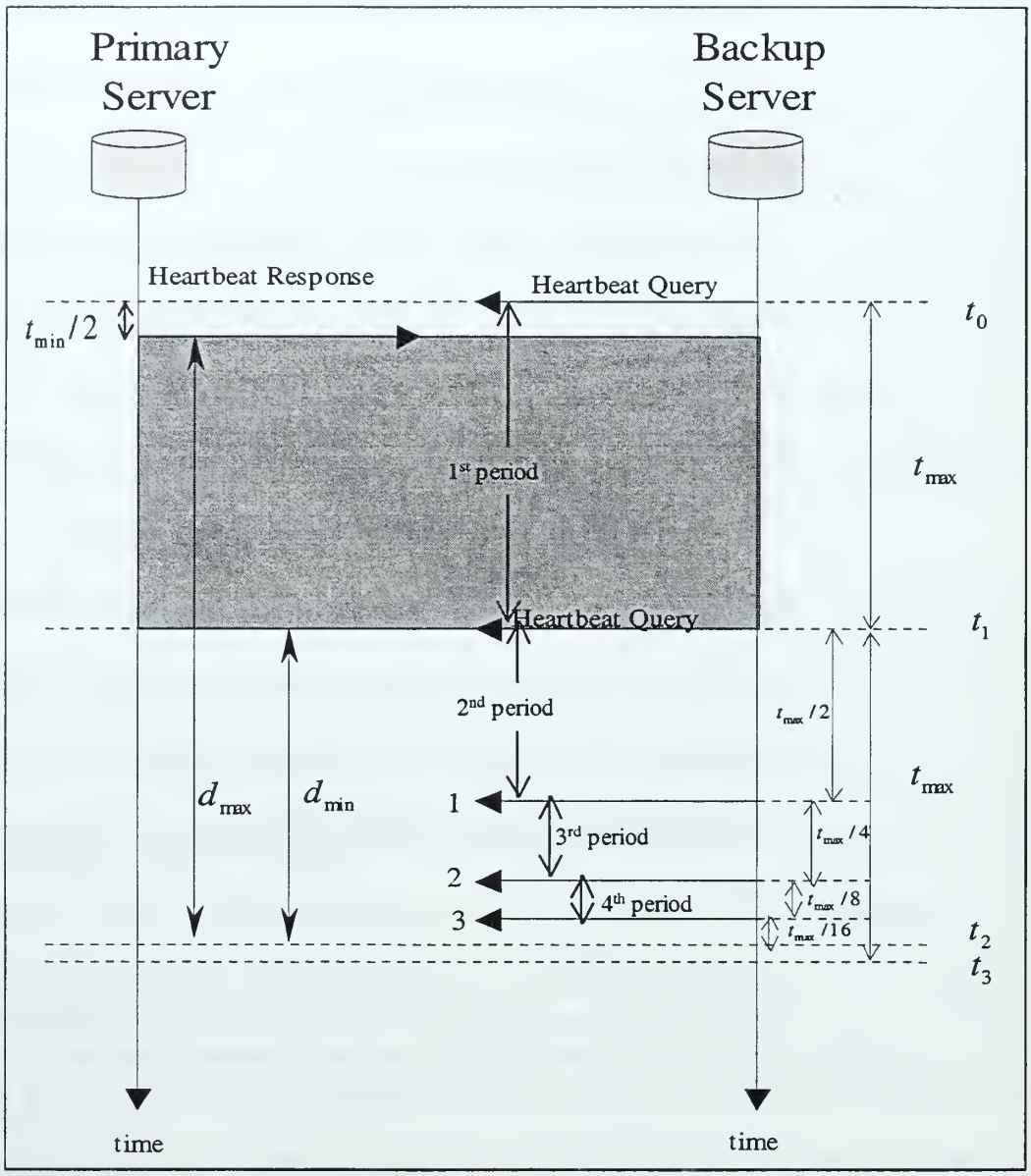


Figure 5.4. Detection Delay in the Accelerated Heartbeat Protocol.

After three consecutive heartbeat message misses, the failure of the primary server is detected (see Figure 5.4) at time t_2. The maximum detection delay time and the minimum detection delay time are shown with the arrows marked by d_{max} and

d_{min}, respectively. Therefore, the relationship between the detection delay, d, and the heartbeat query message interval, t_{max}, is given as follows:

$$\left(\frac{t_{max}}{2}+\frac{t_{max}}{4}+\frac{t_{max}}{8}+\frac{t_{max}}{16}\right)<d<\left(\left(t_{max}-\frac{t_{min}}{2}\right)+\left(\frac{t_{max}}{2}+\frac{t_{max}}{4}+\frac{t_{max}}{8}+\frac{t_{max}}{16}\right)\right) \quad (5.2)$$

$$\frac{15t_{max}}{16}<d<\left(\frac{31t_{max}}{16}-\frac{t_{min}}{2}\right) \quad (5.3)$$

c. Prototyping of the Heartbeat Protocols

The prototypes explained in this section are implemented with the purpose of making a quick performance comparison of the two heartbeat protocols without integrating them with the existing SAAM server source code. Their source code can be found in Appendix A and B.

(1) Constant heartbeat protocol (V.0) prototype: The constant heartbeat protocol prototype includes the following Java class files:

- `PrimaryServer` class
- `PrimaryServerThread` class
- `BackupServer` class
- `TimerHandler` class

The `PrimaryServer` and the `PrimaryServerThread` classes are used for implementing the primary server portion of the constant heartbeat

protocol. On the other hand, the `BackupServer` and the `TimerHandler` classes are used for implementing the backup server portion of the protocol.

The `PrimaryServer` class provides a Graphical User Interface (GUI) to the user (see Figure 5.5). The main components of the GUI are numbered one through five. Specifications and functionalities of these GUI components are summarized in Table 5.1.

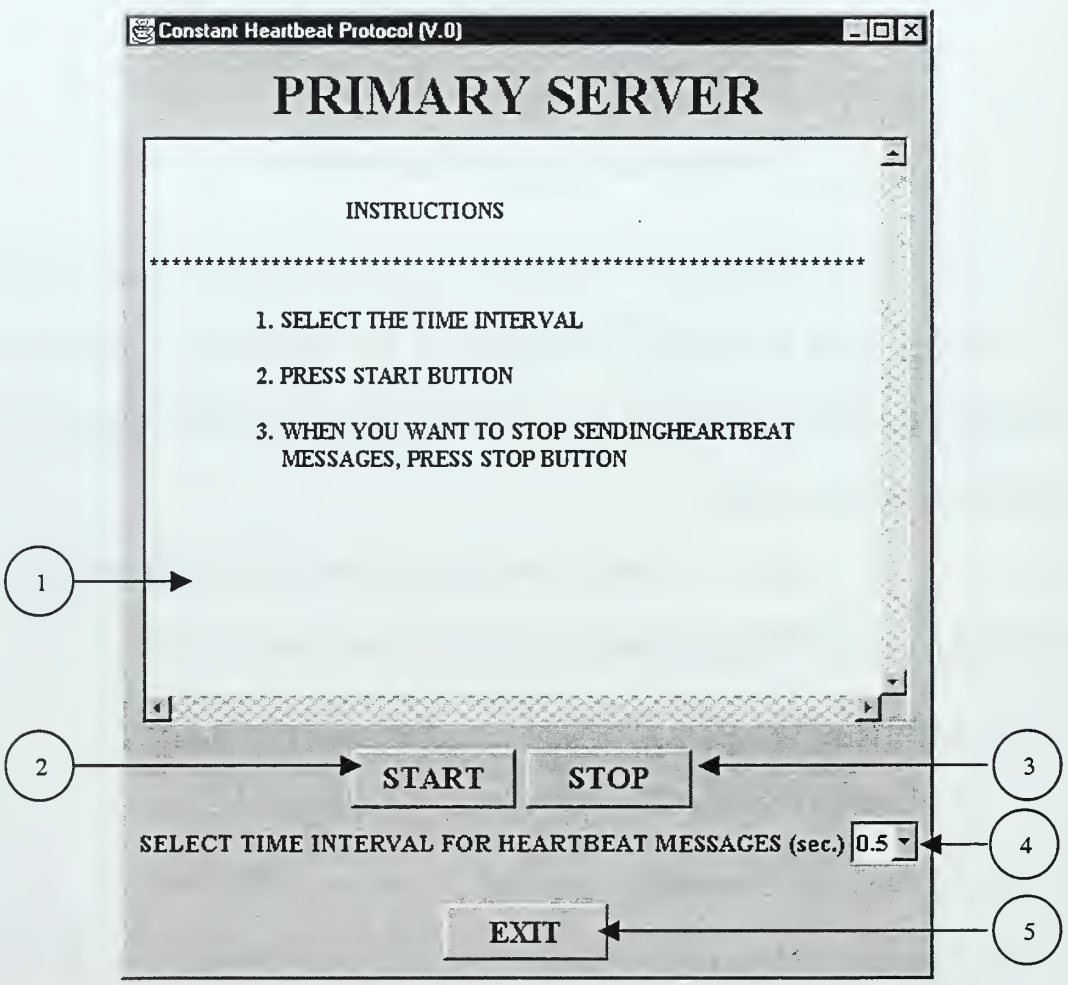


Figure 5.5. The Primary Server GUI of the Constant Heartbeat Protocol.

Number	Name (source code)	Java Object	Functionality
1	display	TextArea	Displays the program instructions and the program outputs to the user.
2	startButton	Button	Initiates the heartbeat message sending process.
3	stopButton	Button	Stops the heartbeat message sending, in order to simulate the failure of the primary server.
4	intervalChoice	Choice	Selects the interval time (in seconds.) between the heartbeat messages (Options are 0.5, 1, 1.5, 2, 2.5 and 3)
5	exitButton	Button	Terminates the program.

Table 5.1. Specifications of the Primary Server GUI Components.

When the user presses the `startButton`, `PrimaryServer` class creates a new thread called `PrimaryServerThread`, which periodically sends the heartbeat messages to the backup server. Unless the `stopButton` is pressed, `PrimaryServerThread` always loops inside in its `run` method. In every loop of the `run` method, the `PrimaryServerThread` first constructs the heartbeat message, then sends the heartbeat message, and finally sleeps for an interval period. As the heartbeat messages are transmitted to the backup server, the messages sent and the times of their departure are displayed in the GUI (see Figure 5.6).

The heartbeat message transmitted by the primary server is implemented as a text string, and consists of four sub-strings. The first sub-string is the string representation of the interval value selected by the user using the intervalChoice menu. The second sub-string is the tilde character ("~") used as a delimiter. The third sub-string is the "I am alive" string. The fourth sub-string is the text representation of the

heartbeat message number. The complete heartbeat messages string are shown in quotes in Figure5.6.

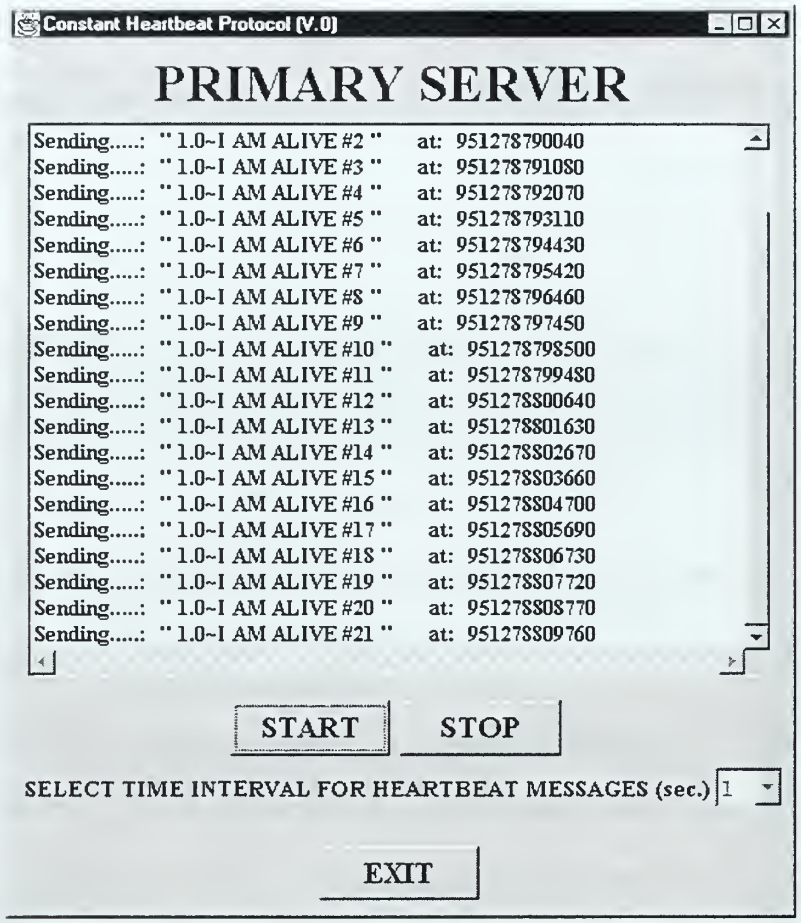

Constant Heartbeat Protocol (V.0)
PRIMARY SERVER
Sending.....: " 1.0~I AM ALIVE #2 " at: 951278790040
Sending.....: " 1.0~I AM ALIVE #3 " at: 951278791080
Sending.....: " 1.0~I AM ALIVE #4 " at: 951278792070
Sending.....: " 1.0~I AM ALIVE #5 " at: 951278793110
Sending.....: " 1.0~I AM ALIVE #6 " at: 951278794430
Sending.....: " 1.0~I AM ALIVE #7 " at: 951278795420
Sending.....: " 1.0~I AM ALIVE #8 " at: 951278796460
Sending.....: " 1.0~I AM ALIVE #9 " at: 951278797450
Sending.....: " 1.0~I AM ALIVE #10 " at: 951278798500
Sending.....: " 1.0~I AM ALIVE #11 " at: 951278799480
Sending.....: " 1.0~I AM ALIVE #12 " at: 951278800640
Sending.....: " 1.0~I AM ALIVE #13 " at: 951278801630
Sending.....: " 1.0~I AM ALIVE #14 " at: 951278802670
Sending.....: " 1.0~I AM ALIVE #15 " at: 951278803660
Sending.....: " 1.0~I AM ALIVE #16 " at: 951278804700
Sending.....: " 1.0~I AM ALIVE #17 " at: 951278805690
Sending.....: " 1.0~I AM ALIVE #18 " at: 951278806730
Sending.....: " 1.0~I AM ALIVE #19 " at: 951278807720
Sending.....: " 1.0~I AM ALIVE #20 " at: 951278808770
Sending.....: " 1.0~I AM ALIVE #21 " at: 951278809760
START
STOP
SELECT TIME INTERVAL FOR HEARTBEAT MESSAGES (sec.) 1
EXIT

Figure 5.6. The Primary Server GUI while Sending Heartbeat Messages.

The *BackupServer* class provides a GUI similar to that of the *PrimaryServer* class (see Figure 5.7). The main components of the GUI are numbered one through four. Specifications and functionalities of these GUI components are summarized in Table 5.2.

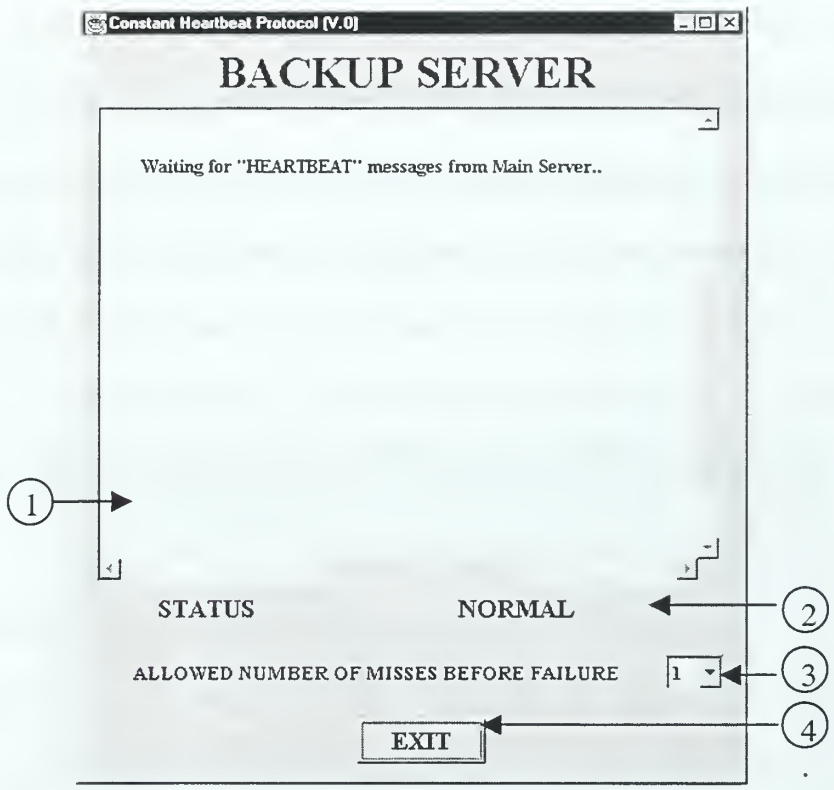


Figure 5.7. The Backup Server GUI of the Constant Heartbeat Protocol.

Number	Name (source code)	Java Object	Functionality
1	display	TextArea	Displays the program outputs to the user.
2	statusDisplayLabel	Label	Shows the primary server's status. If the primary server is running, then its color is green and "NORMAL" is written on it. If the primary server is down, then its color is red and "PRIMARY SEREVR IS DOWN" is written on it.
3	allowedMissChoice	Choice	Selects the allowed-miss value (Options are 1, 2, 3, and 4)
4	exitButton	Button	Terminates the program.

Table 5.2. Specifications of the Backup Server GUI Components.

The backup server listens to the heartbeat messages coming from the primary server via Java `ServerSocket`. Whenever the primary server makes a connection with the backup server to send a heartbeat message, the backup server creates a new Java `Socket` and retrieves the heartbeat message from the input stream. Since the interval value between the heartbeat messages is determined on the primary server side, the backup server is not aware of this value. However, when the backup server receives the first message, the message is tokenized and the heartbeat interval value is retrieved from the message string. The backup server will use this interval value when performing a time check on heartbeat messages.

In order to perform time checks on heartbeat messages, the backup server has a `Timer` object called `timer`. After the first message is received, the `timer` is started with the initial delay of $((allowedMiss+1)\cdot intervalValue)$ seconds. For example, if the allowed-miss is equal to three, and the time interval between heartbeat messages is one second, then the initial delay of the `timer` would be four seconds.

Whenever the backup server receives a new heartbeat message, then the backup server restarts the `timer` with its initial delay. Thus, as long as the primary server continues to send the heartbeat messages, `timer` never expires. However, if the primary server fails to send heartbeat messages, eventually `timer` will expire indicating the failure of the primary server. When the `timer` expires, a Java action event is generated. This action event is heard and handled by the `TimerHandler` class. The generated action causes the execution of the `actionPerformed()` method of the `TimerHandler` class. Declaration of the failure of the primary server is performed in

this `actionPerformed()` method by calling the `setPrimaryServerStatus()` method of the `BackupServer` class.

After the failure of the primary server is detected, the failure detection time, the receive time of the last message, and elapsed time (difference between the receive time of the last message and the failure detection time) are displayed, as shown in Figure 5.8.

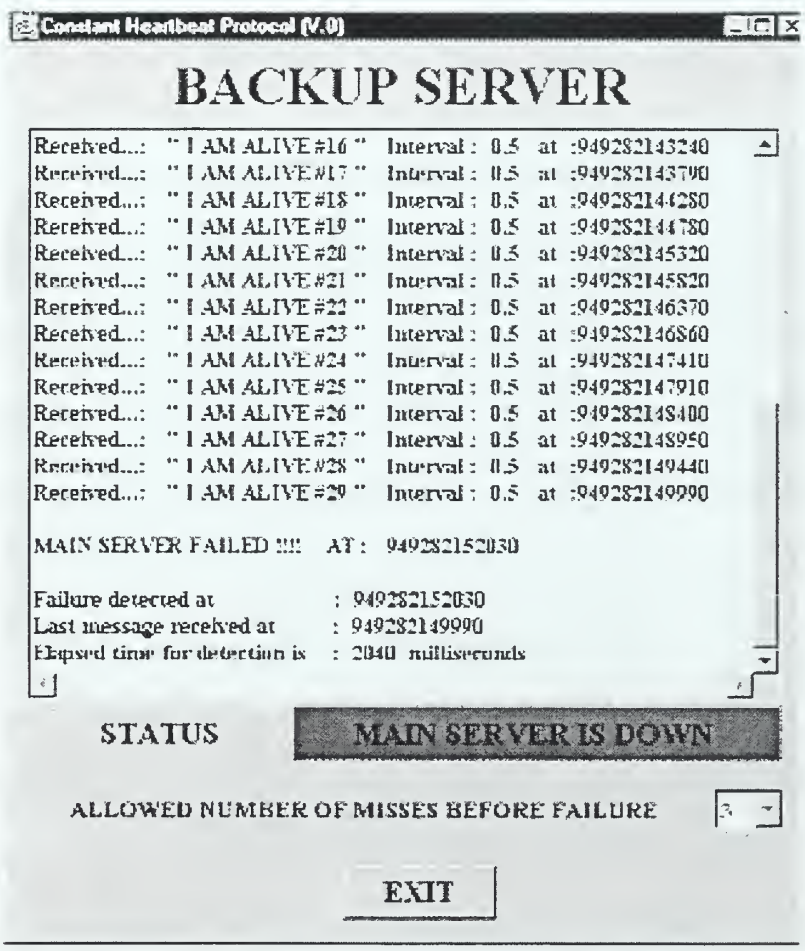

Constant Heartbeat Protocol (V.0)
BACKUP SERVER
Received...: " I AM ALIVE #16 " Interval : 0.5 at :949282143240
Received...: " I AM ALIVE #17 " Interval : 0.5 at :949282143790
Received...: " I AM ALIVE #18 " Interval : 0.5 at :949282144280
Received...: " I AM ALIVE #19 " Interval : 0.5 at :949282144780
Received...: " I AM ALIVE #20 " Interval : 0.5 at :949282145320
Received...: " I AM ALIVE #21 " Interval : 0.5 at :949282145820
Received...: " I AM ALIVE #22 " Interval : 0.5 at :949282146370
Received...: " I AM ALIVE #23 " Interval : 0.5 at :949282146860
Received...: " I AM ALIVE #24 " Interval : 0.5 at :949282147410
Received...: " I AM ALIVE #25 " Interval : 0.5 at :949282147910
Received...: " I AM ALIVE #26 " Interval : 0.5 at :949282148400
Received...: " I AM ALIVE #27 " Interval : 0.5 at :949282148950
Received...: " I AM ALIVE #28 " Interval : 0.5 at :949282149440
Received...: " I AM ALIVE #29 " Interval : 0.5 at :949282149990
MAIN SERVER FAILED !!! AT : 949282152030
Failure detected at : 949282152030
Last message received at : 949282149990
Elapsed time for detection is : 2040 milliseconds
STATUS
MAIN SERVER IS DOWN
ALLOWED NUMBER OF MISSES BEFORE FAILURE
EXIT

Figure 5.8. The Backup Server GUI after the Failure of the Primary Server.

(2) Accelerated heartbeat protocol (V.1) prototype: The following Java classes are used in the implementation of the accelerated heartbeat protocol:

- *PrimaryServer* class
- *PrimaryServerThread* class
- *BackupServer* class
- *MessageTimerHandler* class
- *AckReceiveThread* class
- *AckTimerHandler* class

The *PrimaryServer* and the *PrimaryServerThread* classes are used for implementing the primary server portion of the protocol. The *BackupServer*, the *MessageTimerHandler*, the *AckReceiveThread*, and the *AckTimerHandler* classes are used for implementing the backup server portion of the protocol.

The *PrimaryServer* class provides a GUI (see Figure 5.9) to the user. The main components of the GUI are numbered one through three. Specifications and functionalities of these GUI components are summarized in Table 5.3.

The main responsibility of the *PrimaryServer* class is to listen to the heartbeat query messages coming from the backup server. *PrimaryServer* class listens to the heartbeat query messages via Java *ServerSocket*. Whenever the connection is made by the backup server to send a heartbeat query message, the primary server creates a new Java *Socket* and a new thread called, *PrimaryServerThread* to

handle the connection. The *PrimaryServerThread* class is responsible for retrieving the message string from the socket and sending the corresponding heartbeat response messages. Whenever the *PrimaryServerThread* receives a heartbeat query message, it displays the message arrival time on the GUI. After that, the *PrimaryServerThread* sends the heartbeat response message. The heartbeat response message is a string with the text of, "YES, I AM ALIVE". After the heartbeat response message is sent, *PrimaryServerThread* displays the time that the message has sent on the text area (see Figure 5.9).

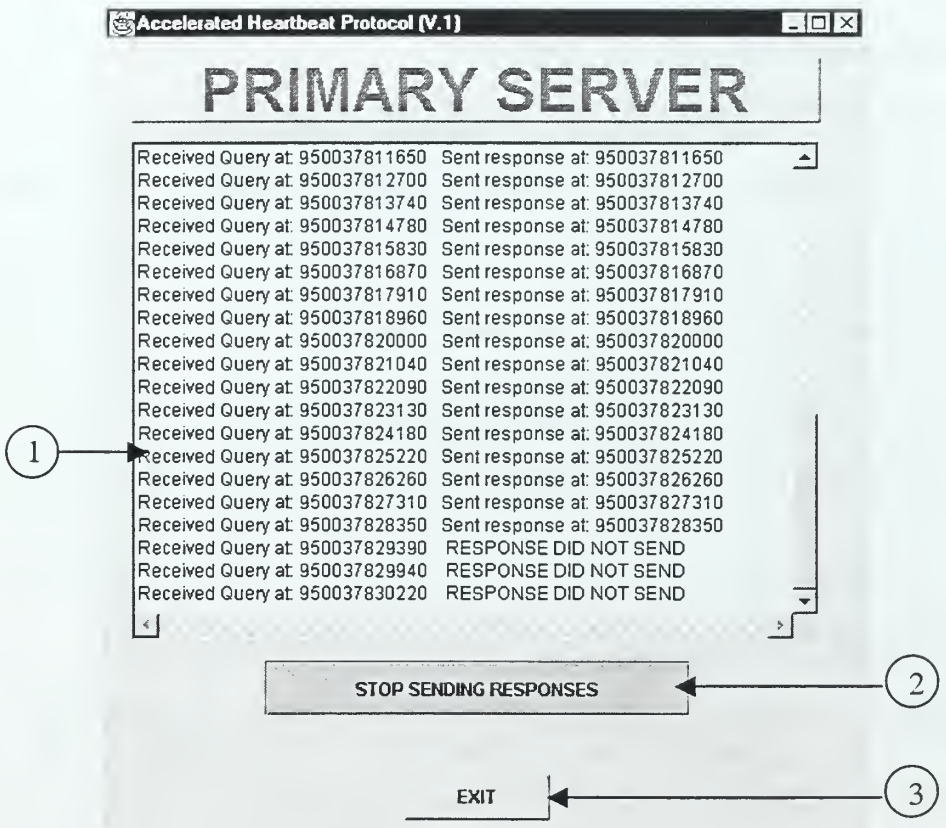

Accelerated Heartbeat Protocol (V.1)
PRIMARY SERVER
Received Query at: 950037811650 Sent response at: 950037811650
Received Query at: 950037812700 Sent response at: 950037812700
Received Query at: 950037813740 Sent response at: 950037813740
Received Query at: 950037814780 Sent response at: 950037814780
Received Query at: 950037815830 Sent response at: 950037815830
Received Query at: 950037816870 Sent response at: 950037816870
Received Query at: 950037817910 Sent response at: 950037817910
Received Query at: 950037818960 Sent response at: 950037818960
Received Query at: 950037820000 Sent response at: 950037820000
Received Query at: 950037821040 Sent response at: 950037821040
Received Query at: 950037822090 Sent response at: 950037822090
Received Query at: 950037823130 Sent response at: 950037823130
Received Query at: 950037824180 Sent response at: 950037824180
Received Query at: 950037825220 Sent response at: 950037825220
Received Query at: 950037826260 Sent response at: 950037826260
Received Query at: 950037827310 Sent response at: 950037827310
Received Query at: 950037828350 Sent response at: 950037828350
Received Query at: 950037829390 RESPONSE DID NOT SEND
Received Query at: 950037829940 RESPONSE DID NOT SEND
Received Query at: 950037830220 RESPONSE DID NOT SEND
1
STOP SENDING RESPONSES
2
EXIT
3

Figure 5.9. The Primary Server GUI of the Accelerated Heartbeat Protocol.

Number	Name (source code)	Java Object	Functionality
1	display	TextArea	Displays the program outputs to the user.
2	toggleButton	JToggleButton	Stops heartbeat response message transmission with the first click and restarts the heartbeat message transmission with the second click.
3	exitButton	JButton	Terminates the program.

Table 5.3. Specifications of the Primary Server GUI Components.

The `BackupServer` class provides a GUI for the backup server portion of the accelerated protocol (see Figure 5.10). The main components of the GUI are numbered one through six. Specifications and functionalities of these GUI components are summarized in Table 5.4.

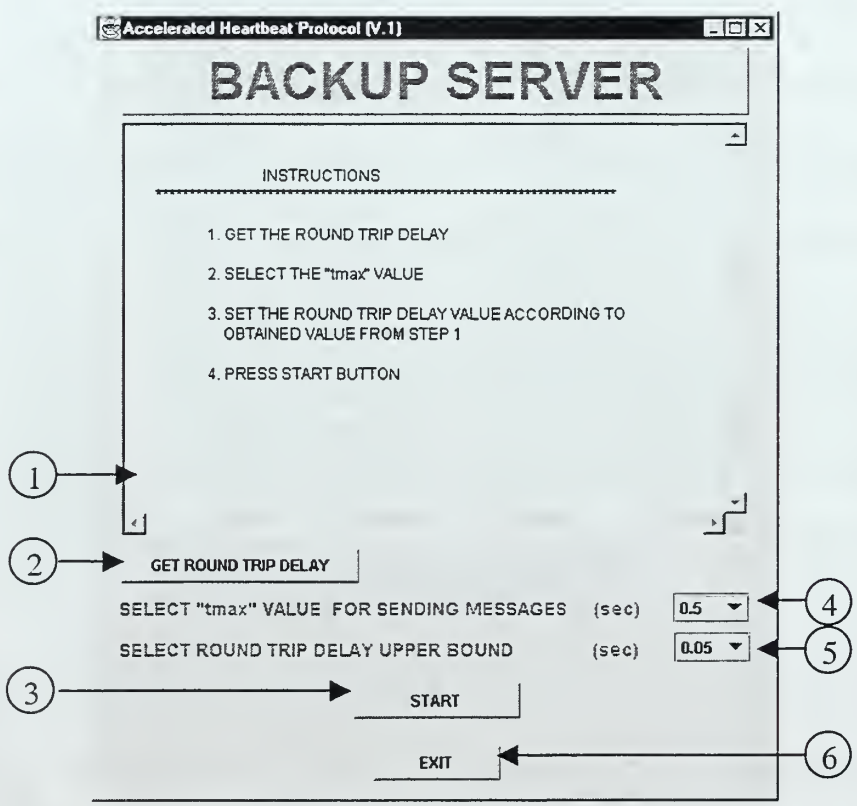


Figure 5.10. The Backup Server GUI of the Accelerated Heartbeat Protocol.

Number	Name (source code)	Java Object	Functionality
1	display	TextArea	Displays the program outputs to the user.
2	rtdButton	JButton	Calculates the round trip delay between two servers by sending four heartbeat queries.
3	startButton	JButton	Starts sending heartbeat query messages
4	tmaxComboBox	JComboBox	Selects the *tmax* value, which is the maximum interval time between heartbeat query messages (Options are 0.5, 1, 1.5, 2, 2.5, 3, 3.5 and 4)
5	roundTripDelayComboBox	JComboBox	Selects the *tmin* value, which is the round trip delay upper bound between two servers (Options are 0.05, 0.06, ……., 0.30)
6	exitButton	JButton	Terminates the program.

Table 5.4. Specifications of the Backup Server GUI Components.

The main responsibility of the `BackupServer` class is to send the heartbeat query messages to the primary server, and to implement the accelerated heartbeat protocol rules explained in Section A.1.b. The heartbeat query messages are sent from the `BackupServer` class. However, the heartbeat response messages are received by the `AckReceiveThread` class. When the start button is pressed, the `BackupServer` class creates two `Timer` objects called `messageSendTimer` and `ackReceiveTimer`.

The purpose of the `messageSendTimer` is to provide periodical heartbeat query message transmission. The `messageSendTimer` is a repeating timer and always restarts with its initial delay, which is set to the t_{max} value. Whenever the `messageSendTimer` expires, the `actionPerformed()` method of the `MessageTimerHandler` class is executed. In this `actionPerformed()` method, the `sendMessage()` method of the `BackupServer` class is called. After that, the

backup server sends a new heartbeat query message to the primary server. For example, if the t_{max} is equals to two, then every two seconds the backup server sends a new heartbeat query message.

The purpose of the `ackReceiveTimer` is to implement time checks on the heartbeat response messages. The `ackReceiveTimer` is started when the first heartbeat query message is sent. Its value is set to half of the current interval. If the `AckReceiveThread` receives the heartbeat response message prior to the expiration of the `ackReceiveTimer`, then the `ackReceiveTimer` is stopped before its expiration. Therefore, as long as the heartbeat response messages arrive regularly within the first half of the current interval, the `ackReceiveTimer` never expires.

However, if the backup server does not receive a heartbeat response message within the first half of the current interval, then the `ackReceiveTimer` expires and causes the execution of the `actionPerformed()` method of the `AckTimerHandler` class. In this method, the `messageSendTimer` is stopped, a new heartbeat query message is sent, and the current interval value is set to half of the previous interval value. If the new interval value ever becomes less than the round trip delay upper bound value, then failure of the primary server is declared (see Figure 5.11). However, if the backup server receives a heartbeat response message before determining the status of the primary server, then the message send timer is restarted and the current interval is set back to its original value, t_{max}.

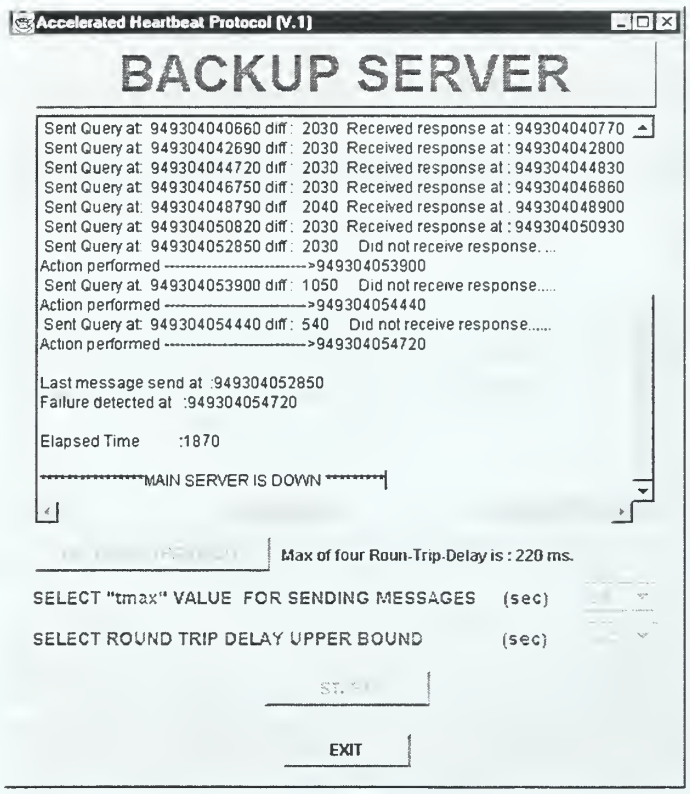

Accelerated Heartbeat Protocol (V.1)
BACKUP SERVER
Sent Query at: 949304040660 diff : 2030 Received response at : 949304040770
Sent Query at: 949304042690 diff : 2030 Received response at : 949304042800
Sent Query at: 949304044720 diff : 2030 Received response at : 949304044830
Sent Query at: 949304046750 diff : 2030 Received response at : 949304046860
Sent Query at: 949304048790 diff : 2040 Received response at : 949304048900
Sent Query at: 949304050820 diff : 2030 Received response at : 949304050930
Sent Query at: 949304052850 diff : 2030 Did not receive response. ...
Action performed ----------->949304053900
Sent Query at: 949304053900 diff : 1050 Did not receive response.....
Action performed ----------->949304054440
Sent Query at: 949304054440 diff : 540 Did not receive response.....
Action performed ----------->949304054720
Last message send at :949304052850
Failure detected at :949304054720
Elapsed Time :1870
MAIN SERVER IS DOWN
Max of four Roun-Trip-Delay is : 220 ms.
SELECT "tmax" VALUE FOR SENDING MESSAGES (sec)
SELECT ROUND TRIP DELAY UPPER BOUND (sec)
EXIT

Figure 5.11. The Backup Server GUI After Failure.

d. Performance Comparison of the Heartbeat Protocols

In this section, the failure detection delays and other performance results of the two heartbeat protocols are compared. In order to avoid the unexpected network delay, our tests are performed by running both the primary server and the backup server programs on the same computer. (The processor of the test computer was a 233 MHz. Pentium II.)

First, the performance of the constant heartbeat protocol is evaluated. A set of tests are performed using different interval values between the heartbeat messages. In this test, the allowed-miss values of one, two, three, and four are tested and the failure detection delays are calculated while keeping the allowed-misses constant (the results are

summarized in Figure 5.12). The failure detection delays are based on the maximum delay (illustrated with d_{max} in Figure 5.3), and calculated by subtracting the receive time of the last heartbeat message from the failure detection time.

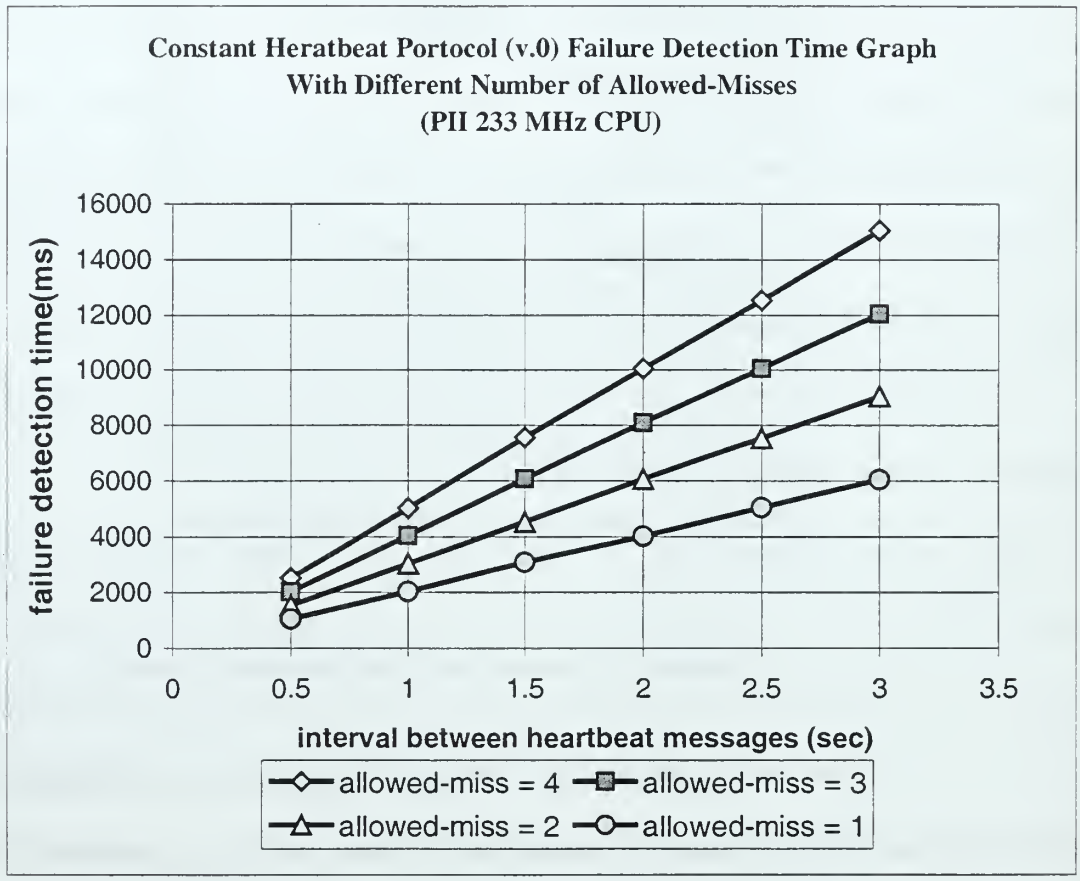


Figure 5.12. The Constant Heartbeat Protocol Failure Detection Delay Graph.

Second, the performance of the accelerated heartbeat protocol is evaluated. In order to test certain allowed-misses (similar to the constant heartbeat protocol), for each interval (t_{max}) value, the round trip delay upper bounds are set to specific values so that only desired number of heartbeat response message misses are allowed. A set of tests are performed by using different regular interval values between

the heartbeat query messages for the same allowed-miss values. In this test, allowed-miss values of one, two, three, and four are tested, and failure detection delays are calculated (the results are summarized in Figure 5.13). The failure detection delays are based on the maximum delay (illustrated with d_{max} in Figure 5.4), and are calculated by subtracting the receive time of the last message from the failure detection time.

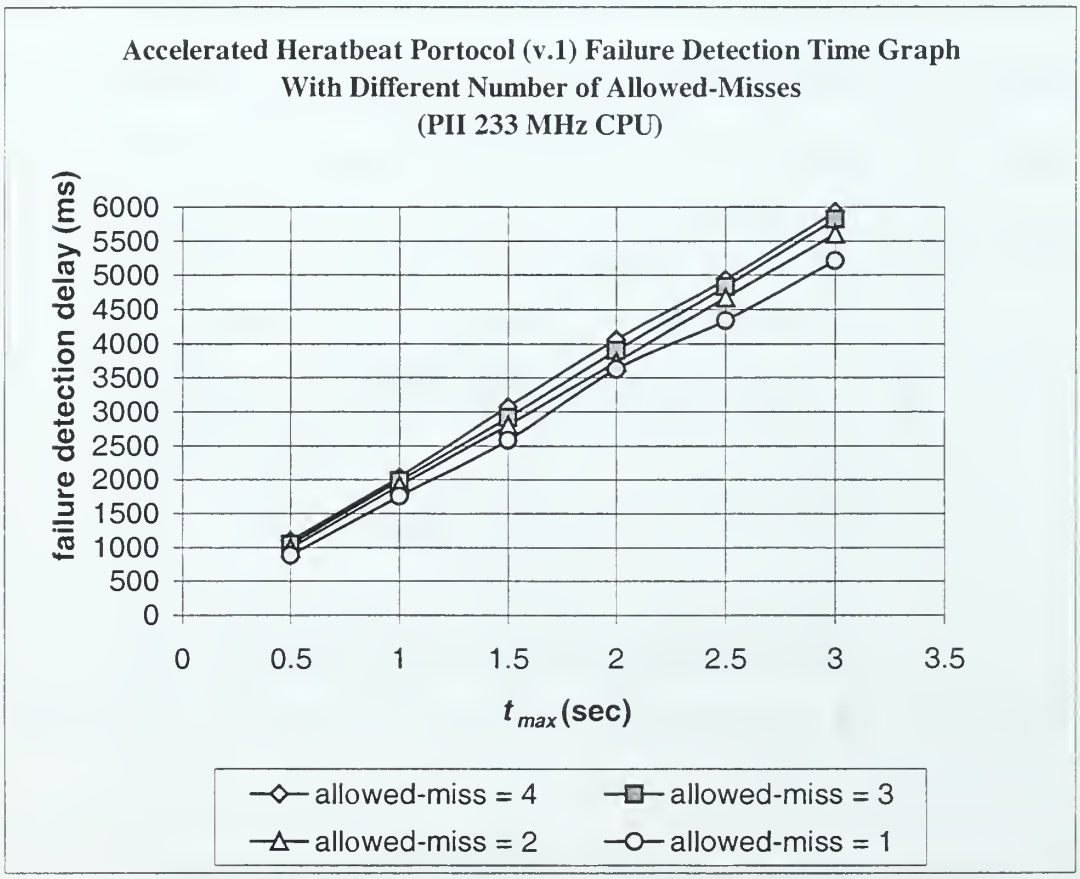


Figure 5.13. The Accelerated Heartbeat Protocol Detection Delay Graph.

Third, for each tested interval (t_{max}) values, the failure detection delays of the constant heartbeat protocol and the accelerated heartbeat protocol are compared for different allowed-misses. Comparisons are made for the interval values of 0.5, 1.0, 1.5,

2.0, 2.5, and 3.0, and the results are provided in Figures 5.14, 5.15, 5.16, 5.17, 5.18 and 5.19, respectively.

When the interval between regular heartbeat messages is set to a half-second, the lowest failure detection delay offered by the constant heartbeat protocol is 1052.5 milliseconds, and only one heartbeat message miss is tolerated (see Figure 5.14). On the other hand, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 680 milliseconds. Additionally, although the constant heartbeat protocol can only allow one message miss to detect the failure in 1052.5 milliseconds, the accelerated heartbeat protocol can allow three message misses within almost the same detection delay period (1040 ms.) (see Figure 5.14).

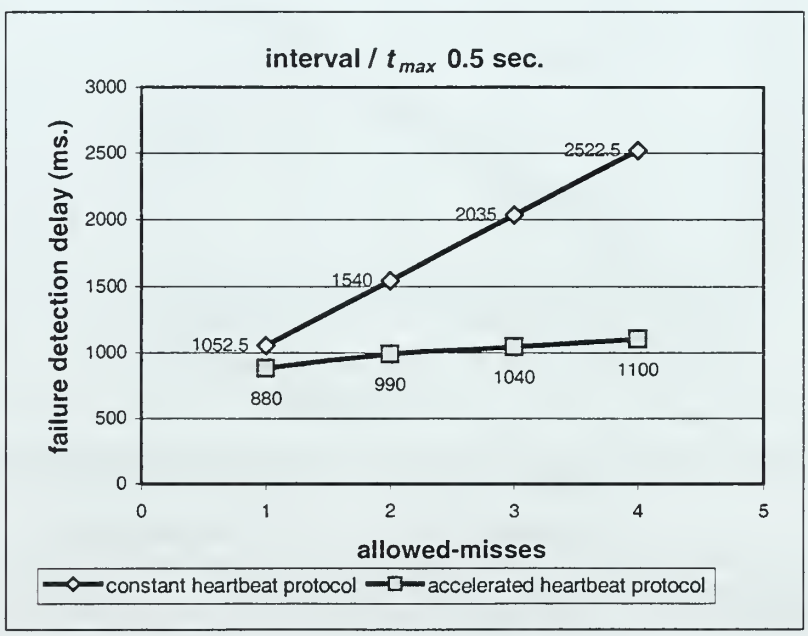


Figure 5.14. Comparison of the Protocols When the Interval (t_{max}) is 0.5 Seconds.

When the interval between heartbeat messages is set to one second, the lowest failure detection delay offered by the constant heartbeat protocol is 2032.5

milliseconds, and only one heartbeat message miss is tolerated (see Figure 5.15). Alternately, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 1760 milliseconds. Additionally, although the constant heartbeat protocol can only allow one message miss to detect the failure in 2032.5 milliseconds, the accelerated heartbeat protocol can allow four message misses within almost the same detection delay period (2040 ms.) (see Figure 5.15).

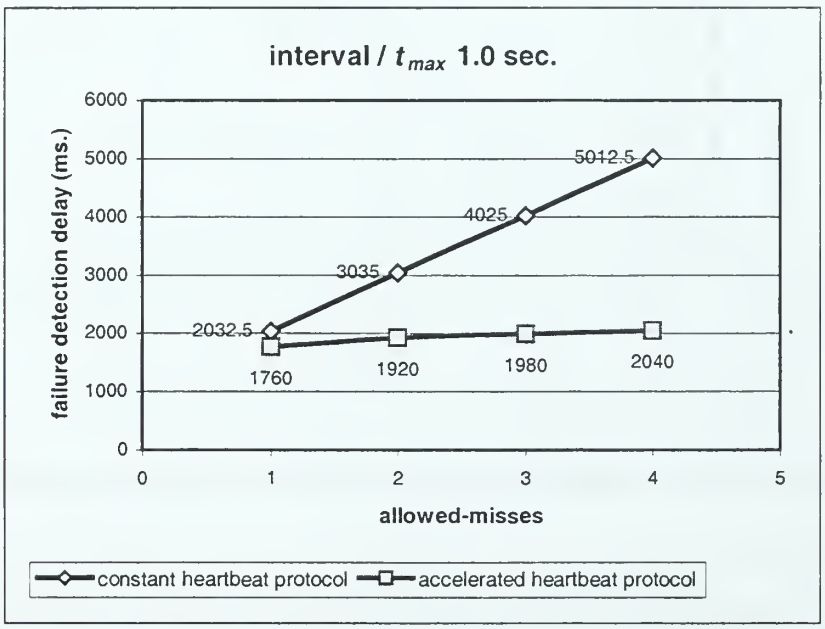


Figure 5.15. Comparison of the Protocols When the Interval (t_{max}) is 1 Second.

When the interval between heartbeat messages is set to one and a half seconds, the lowest failure detection delay offered by the constant heartbeat protocol is 3075 milliseconds, and only one heartbeat message miss is tolerated (see Figure 5.16). On the other hand, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 2580 milliseconds. Additionally,

although the constant heartbeat protocol can only allow one message miss to detect the failure in 3075 milliseconds, the accelerated heartbeat protocol can allow four message misses within almost the same detection delay period (3080 ms.)(See Figure 5.16).

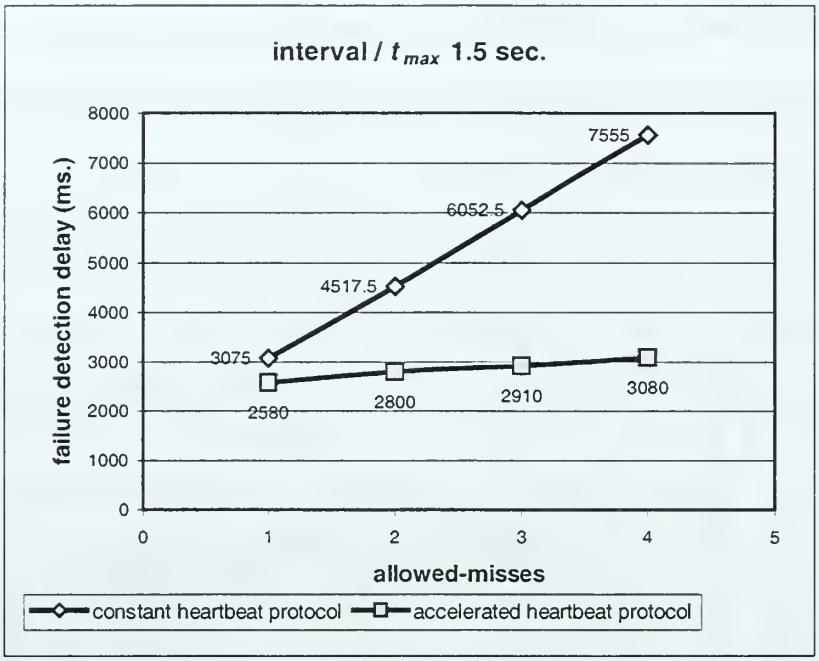


Figure 5.16. Comparison of the Protocols When the Interval (t_{max}) is 1.5 Seconds.

When the interval between heartbeat messages is set to two seconds, the lowest failure detection delay offered by the constant heartbeat protocol is 4022.5 milliseconds and only one heartbeat message miss is tolerated (see Figure 5.17). On the other hand, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 3630 milliseconds. Additionally, although the constant heartbeat protocol can only allow one message miss to detect the failure in 4022.5 milliseconds, the accelerated heartbeat protocol can allow four message misses within almost the same detection delay period (4060 ms.) (see Figure 5.17).

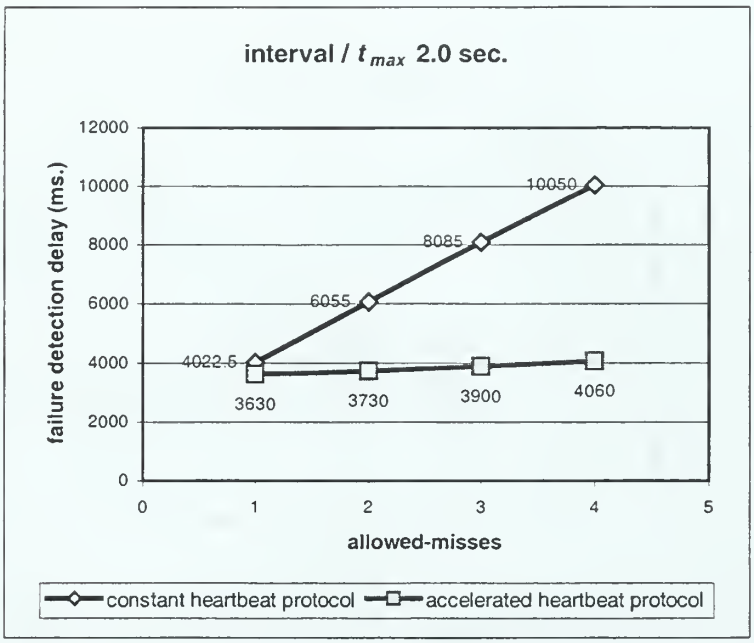


Figure 5.17. Comparison of the Protocols When the Interval (t_{max}) is 2.0 Seconds.

When the interval between heartbeat messages is set to two and a half-second, the lowest failure detection delay offered by the constant heartbeat protocol is 5037.5 milliseconds and only one heartbeat message miss is tolerated (see Figure 5.18). On the other hand, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 4340 milliseconds. Additionally, although the constant heartbeat protocol can only allow one message miss to detect the failure in 5037.5 milliseconds, the accelerated heartbeat protocol can allow four message misses within even lower detection delay period of 4940 ms. (see Figure 5.18).

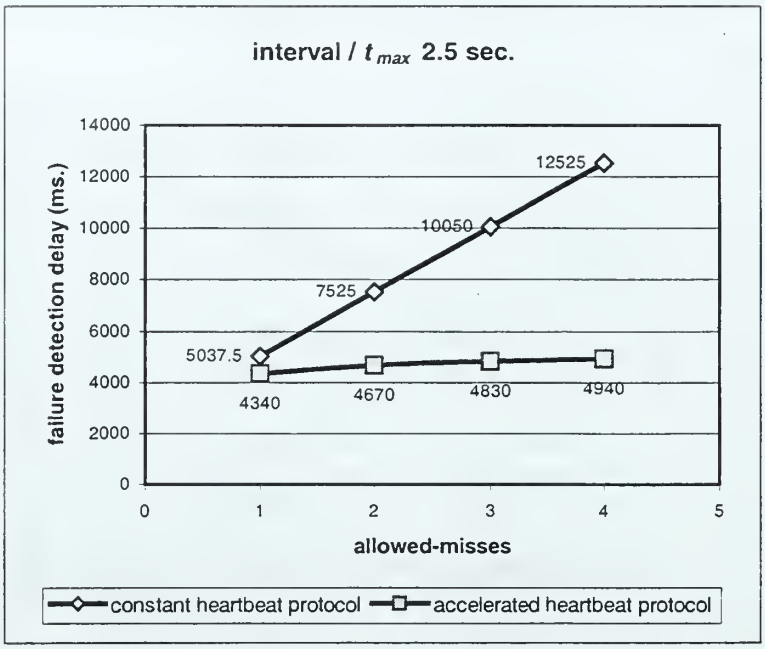


Figure 5.18. Comparison of the Protocols When the Interval (t_{max}) is 2.5 Seconds.

When the interval between heartbeat messages is set to three seconds, the lowest failure detection delay offered by the constant heartbeat protocol is 6040 milliseconds and only one heartbeat message miss is tolerated (see Figure 5.19). On the other hand, when only one heartbeat response message miss is allowed, the accelerated heartbeat protocol can detect the failure in 5210 milliseconds. Additionally, although the constant heartbeat protocol can only allow one message miss to detect the failure in 6040 milliseconds, the accelerated heartbeat protocol can allow four message misses within even lower detection delay period of 5930 ms. (see Figure 5.19).

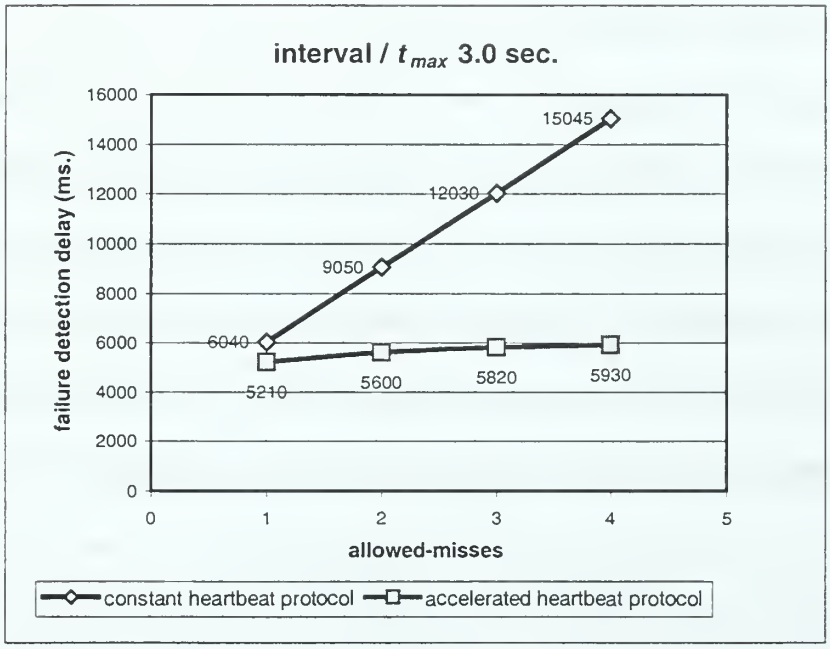


Figure 5.19. Comparison of the Protocols When the Interval (t_{max}) is 3.0 Seconds.

The rate at which heartbeat messages are sent in the protocol should be small in order to reduce protocol overhead. To compare the protocol overhead of the two heartbeat protocols, consider the case that the failure detection is required to happen in two seconds and only three consecutive message misses are allowed. In this case, constant heartbeat protocol interval value should be set to 0.5 seconds (see Figure 5.12). Consequently, the constant heartbeat protocol will introduce two messages per second. Under the same condition, the accelerated heartbeat protocol t_{max} value should be set to one second (see Figure 5.13). Consequently, during the normal operation of the primary server the accelerated heartbeat protocol will also introduce two messages (one query and one response) per second. When the primary server fails, the accelerated heartbeat

protocol will introduce three additional query messages. However, in the long run these three messages are negligible. As a result, the two protocols introduce almost the same amount of traffic overhead to the network.

According to the test results explained above, accelerated heartbeat protocol provides better error detection capability than the constant heartbeat protocol. Although both protocols introduce almost the same amount of traffic overhead to the network, the probability of false failure detection in the accelerated heartbeat protocol is much less than that of the constant heartbeat protocol. For these reasons, the accelerated heartbeat protocol will be used as the error detection mechanism for the SAAM server.

e. Preventing False Failure Detection

False failure detection refers to a declaration of the failure of the primary server when the primary server has not actually failed. False failure detection causes both servers to be simultaneously in the active running state. Consequently, routers receiving updates simultaneously from both servers wouldn't be able to determine which update to accept. If the accelerated heartbeat protocol is used as the failure detection mechanism, false failure detection can only occur when the backup server misses more than n (typically n is three or four) consecutive heartbeat response messages due to the loss of either the heartbeat query messages or the heartbeat responses in the network.

Heartbeat messages are communicated in the control channel. Since the control channel has the highest priority among all network traffic in the SAAM region, the loss of heartbeat messages, especially the loss of three or four consecutive heartbeat messages, is unlikely. However, a network failure (i.e., a link failure or a router failure

preventing the communication of two servers) may prevent the communications between the two servers, causing heartbeat message losses.

Failure of a router or a link can have different effects on the failure detection mechanism according to their locations on the network. To illustrate this point, consider a sample network topology given in Figure 5.20. In this sample network topology, the heartbeat messages between the primary and the backup server travel through routers A3, A5, A4, and A0. Therefore, failure of the routers A1, A2, and A6 does not affect the operation of the accelerated heartbeat protocol. However, a failure of the routers A3, A5, A4, and A0 will definitely affect the heartbeat message communication between the two servers. Due to the their positions in the network, a failure of the routers that directly connect to the servers (in this example, A3 or A0) and a failure of the routers that are located on the way of the control channel (in this example, A5 and A4) introduce different effects. Therefore, according to the router locations, the proposed solutions are discussed separately.

A failure in the routers directly connected to the servers will definitely result in false failure detection. In the sample topology, when router A3 fails, the backup server will not receive any heartbeat response messages from the primary server, even though the primary server is still alive. Therefore, the backup server eventually will declare the failure of the primary server. Although there are now two actively running servers, from a router's perspective, as long as the router A3 is not repaired, only the backup server exists. When router A3 is repaired and reinstated, there will be two servers running at the same time.

In order to avoid this co-existence of two active servers, if a router directly connected to the primary server fails, then the primary server must enter the repaired state. The primary server can use the SAAM auto-configuration mechanism to detect the failure of the router that is directly connected to itself. If the primary server sends a DCM message and sees no routers, then the primary server enters the failed state. The same conditions also apply to the backup server. If the backup server sends a DCM message and sees no routers, then it enters the failed state.

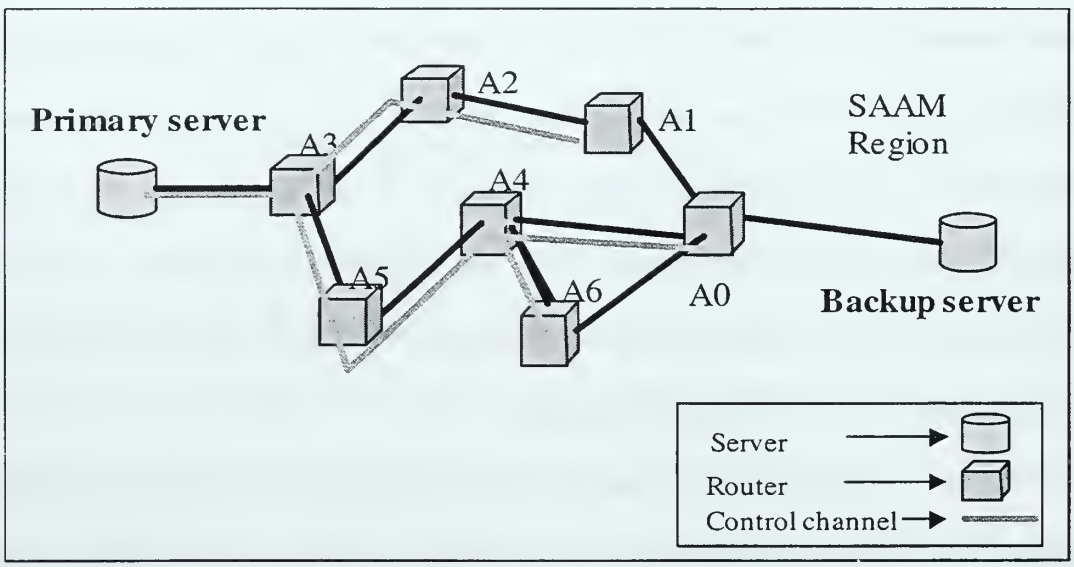


Figure 5.20. An Example Network Topology for the SAAM Region.

A failure of one of the routers that are part of the control channel between the two servers (in this example, A5 and A4) may also result in false failure detection. However, false failure detection caused by this type of router failures can be avoided. In order to avoid false failure detection, it is essential that the auto-configuration mechanism of the SAAM server rebuild the control channel prior to the false failure declaration. By

doing so, heartbeat response messages would be able to reach the backup server by using the newly built control channel, preventing the backup server from making false failure detection. In order to achieve this, proper relationship between the DCM message interval, denoted by C, and the heartbeat query message interval, $t_{\max}$, must be established.

A router can fail at any time. However, the worst case is when a router failure occurs right after the completion of the last auto-configuration cycle, but just before a heartbeat query message departure, (as shown in Figure 5.21). In this case, the current and subsequent heartbeat query messages would not reach the primary server. Consequently, the backup server would not receive any response until t_2. The backup server would declare the failure of the main server at some time before t_2.

In order to avoid false failure detection, the control channel auto-configuration must be performed, at the latest, just before the last heartbeat query message, as shown in Figure 5.21. By this reasoning, if we assume that the round trip delay upper bound allows only three heartbeat response message misses, then the relation between the DCM message interval, C, and the regular heartbeat query interval, $t_{\max}$, must satisfy:

$$\left(\frac{t_{\max}}{2}+\frac{t_{\max}}{4}+\frac{t_{\max}}{8}\right)\geq C \tag{5.4}$$

$$t_{\max}\geq\frac{8C}{7} \tag{5.5}$$

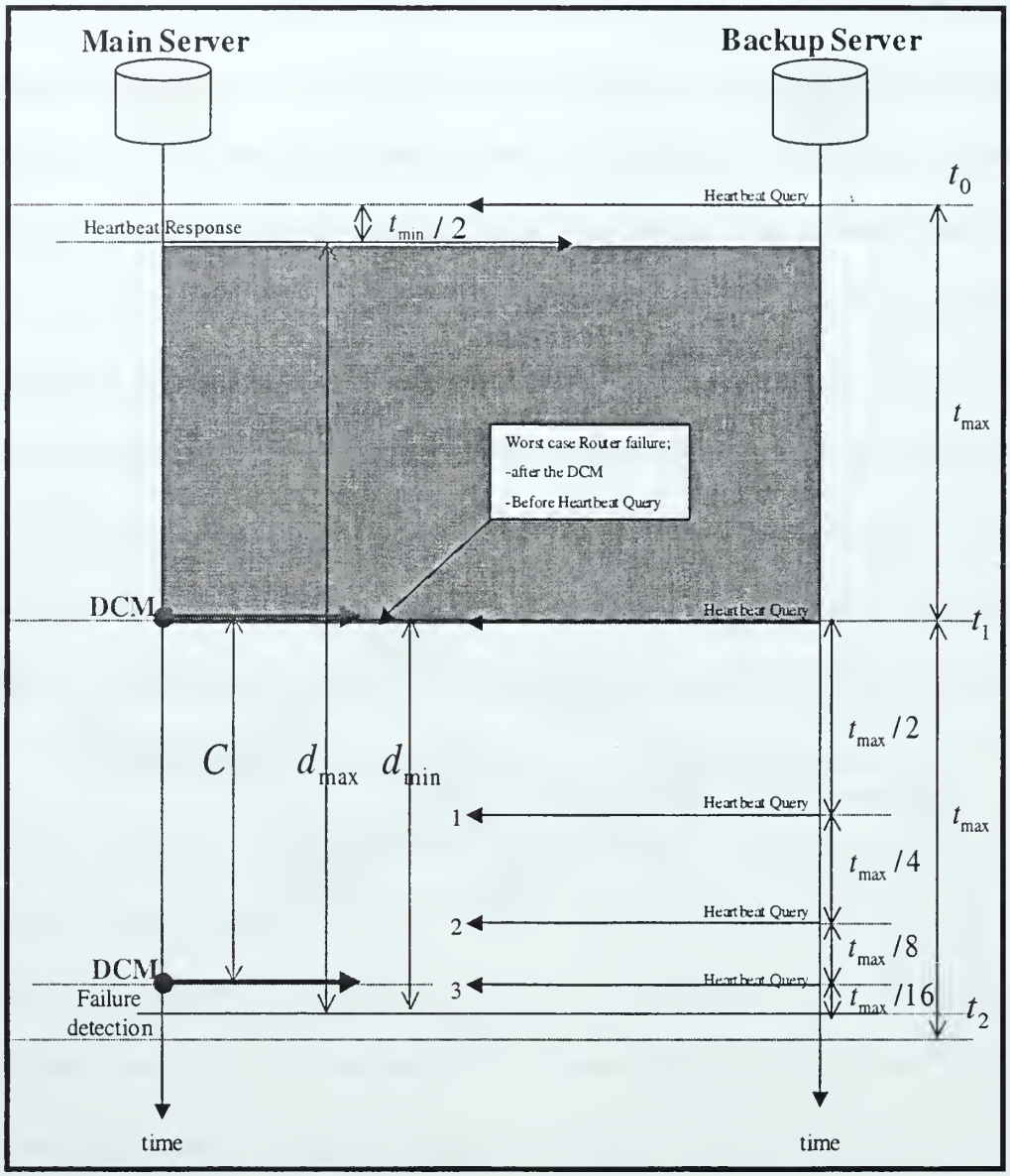


Figure 5.21. Relationship Between DCM and Heartbeat Query Messages.

When the relationship given by Equation 5.5 is provided between the t_{max} and the C values, the auto-configuration mechanism would reconstruct the control channel before false failure detection. Therefore, a false failure detection due to a failure

of one of the routers that are part of the control channel between the two servers will be avoided.

Another preventive measure for false failure detection is to use each DCM message as an unsolicited heartbeat message. Since the DCM messages are sent more frequently than the heartbeat query messages, the backup server always performs an additional check for new DCM messages before declaring the failure of the primary server. Even though the backup server does not receive heartbeat responses, if the backup server receives a new DCM message from the primary server during the current period, it sets the next period to t_{max}.

f. Preventing Late Failure Detection

According to the accelerated heartbeat protocol rules, if in a period, the backup SAAM server sends a heartbeat query message to the primary SAAM server and receives a heartbeat response message within the first half of the current period, then the backup SAAM server makes the length of the next period t_{max} (regardless of the length of the current period). However, if the backup server receives a delayed heartbeat response message sent prior to the primary server's failure but before the event of failure detection, then the detection delay will dramatically increase.

To address this late failure detection problem, the heartbeat messages will be numbered. Specifically, the backup server assigns a different number to each heartbeat query message and records the last message number. The primary server assigns its heartbeat response message number to the last received heartbeat query message number. Finally, the backup server compares the recorded number with the last received heartbeat

response message number. If the numbers match, then the backup server accepts the heartbeat response; otherwise it ignores the heartbeat response message.

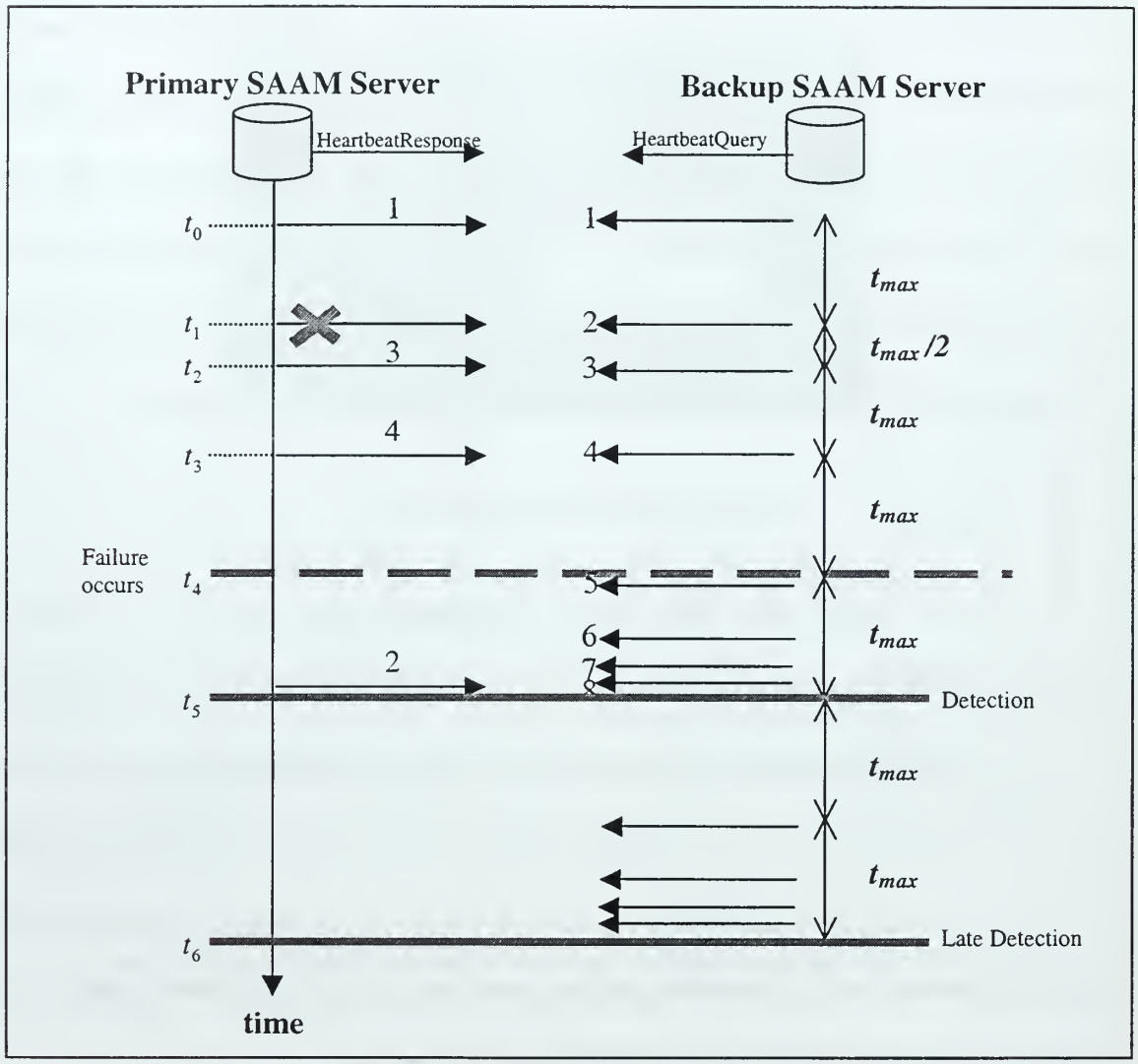


Figure 5.22. Benefits of Heartbeat Message Numbering.

In Figure 5.22, a sample scenario is given to illustrate the benefits of numbering the heartbeat messages. In this example, the primary SAAM server fails at t_4. After the failure of the primary server, the backup server receives a delayed heartbeat

response message just before t_5. If the numbering mechanism is implemented, when the backup server receives heartbeat response message number two, then it ignores this heartbeat response message and detects the failure at t_5. However, if the numbering mechanism is not used, then the backup server thinks that the primary server is still alive, and makes the next period t_{max} at t_5. Therefore, the backup server cannot detect the failure until t_6.

g. Existence of Two Active Servers at the Same Time

The measures described above dramatically reduce the probability of two servers being in the active running state at the same time. However, the probability of two servers being in the active running state is not zero. Therefore, such a case must be considered and handled by the system. In the event of having two active servers at the same time, routers may receive messages (e.g., flow response and flow routing table entry update messages) from both servers and must determine which one to use.

To prevent a situation in which two servers actively run at the same time, one might implement an *explicit server identification scheme*. The explicit server identification scheme refers to sending a special message (e.g., *primary server id message*) from the server entering the active running state to both the routers (including currently active server) to inform them of the identity of the new primary server. Routers receiving updates from two active servers will only accept the updates stamped with the right primary server id. An active server, upon receiving a new primary server identification message shall return to the silent running state.

3. Damage Confinement and Assessment

Before starting the error correction process, it is essential to identify the boundaries of the damage caused by the failure of the primary SAAM server. Although failure detection happens very quickly, there might still be some losses of messages between the routers and the primary SAAM server in the failure detection period. This section discusses the complications introduced by such losses.

The messages communicated among the SAAM components and their destinations are illustrated in Figure 5.23. The effects of heartbeat query and response message losses have been covered earlier in the chapter. Therefore they are not discussed in this section.

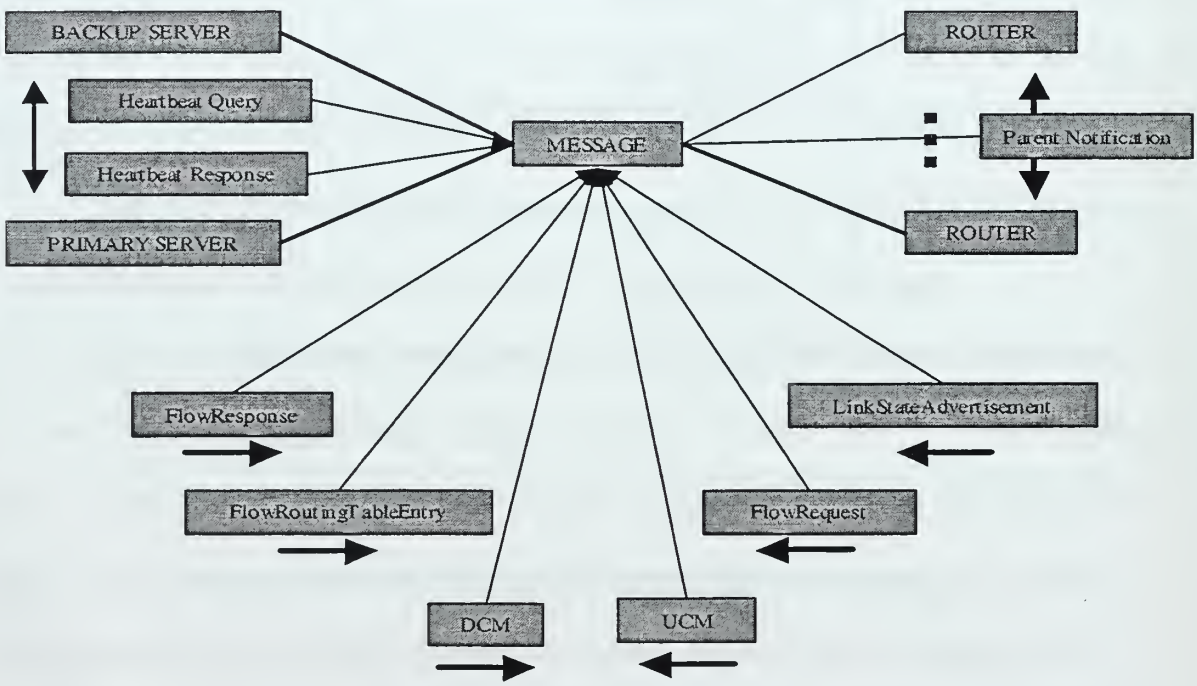


Figure 5.23. Messages Communicated in SAAM.

Among those messages shown in Figure 5.23, the downward configuration message (DCM), upward configuration message (UCM), and parent notification message are used for the SAAM auto-configuration of the control channel. The control channel is used for SAAM protocol specific communications among SAAM components. With SAAM's soft state approach for control channel auto-configuration, both the primary and the backup servers send DCM messages periodically (typically every two seconds) to handle network configuration changes. At the end of each configuration cycle, both servers receive aggregated UCM messages from the routers and construct a new control channel tree. Since the backup server also maintains its control channel by periodically sending separate DCM messages, loss of any auto-configuration related messages during the failure detection delay period would not introduce any problem in the SAAM system.

The Link State Advertisement (LSA) messages are sent from routers to both the primary and the backup servers to inform the servers about the current performance of each service level pipe of each interface of each router. The LSA messages are not sent to servers individually. They are piggybacked to UCM messages at each router in every auto-configuration cycle. Since the backup server always receives the LSA messages in parallel with the primary server, the loss of LSA messages during the failure detection delay period would not introduce any problem in the SAAM system.

The last three message types, flow request, flow routing table entry, and flow response, are used for flow resource reservation in SAAM. Each flow request message is sent to both the primary and the backup servers by some application requesting a QoS flow. However, only the primary server responds to this request. If the primary server determines that the flow request can be supported, then it assigns a flow identifier to the

new flow and allocates a path to it. Before responding to the application, the primary server sends a flow routing table entry to each router in the selected path. Specifically, a flow routing table entry message contains a flow I.D. and the appropriate next hop and service level upon which to forward the packets carrying that flow id. After delivering the flow routing table entry messages, the primary server sends a flow response message to the requesting application. The flow response message simply indicates to the application which flow I.D. it should used for its packets.

During the failure detection period, if a flow request message is lost between the application and the primary server, then there will not be any flow routing table entry message or flow response message coming out from the primary server for this particular flow request. However, after detecting the failure of the primary server, if the backup server enters the active running state before processing the request in its silent running state, then the backup server would be able to send the flow routing table entry and the flow response messages. If this is not the case, then the application needs to repeat its flow request. By the same token, in case of either the flow routing table entry message or the flow response message loss, the requesting application needs to repeat its flow request.

As a result, in the designed model, the failure of the primary server does not cause any major damage to the SAAM architecture; at worst the application will be required to repeat its flow request.

4. Failure Recovery

After a failure of the primary SAAM server is detected and the damage is assessed, the failure should be recovered. In the remote area fault tolerance model for the SAAM server, the recovery process is implemented by using a process called error masking. Error masking is a form of effective error processing and mainly focuses on the corrections made after an error has taken affect.

In general, to mask an error, the erroneous state of the system must contain enough redundancy to enable the delivery of an error-free service from the erroneous internal state. In the SAAM architecture, the redundant backup SAAM server is the one who is responsible for masking the failure of the primary SAAM server. The quality of the services provided to the routers by the primary SAAM server primarily depends upon the accuracy of PIB. Therefore, the backup SAAM server needs to have an up-to-date PIB to mask the failure of the primary SAAM server.

In order to provide an up-to-date PIB on the backup SAAM server, two approaches were considered:

Approach 1. Periodical transfer of PIB contents from the primary SAAM server to the backup SAAM server.

Approach 2. Duplicating and sending all router originated control messages to both the primary SAAM server and the backup SAAM server.

The feasibility of Approach 1 depends on largely how big the PIB data record is. Therefore, in the following sections, the size of the PIB is evaluated and pros and cons of the two approaches are discussed.

a. Size of PIB Data Records

The PIB is currently implemented by using a Java Class Objects. It is stored in the volatile memory (main memory) of the server computer. The PIB consists of three Java objects; `nodes`, `paths` and `links`.

The `nodes` object is a hash table that uses the assigned `node id` of each router as the key (See Figure 5.24). The elements stored in this hash table are references to other hashtables that maintain interface information. The interface hash table uses the `IPv6 address` of each interface as the key. The elements stored in this hashtable are vectors called `slps`. An `slps` vector stores objects that describe the characteristics of a service level pipe. [Ref. 36]

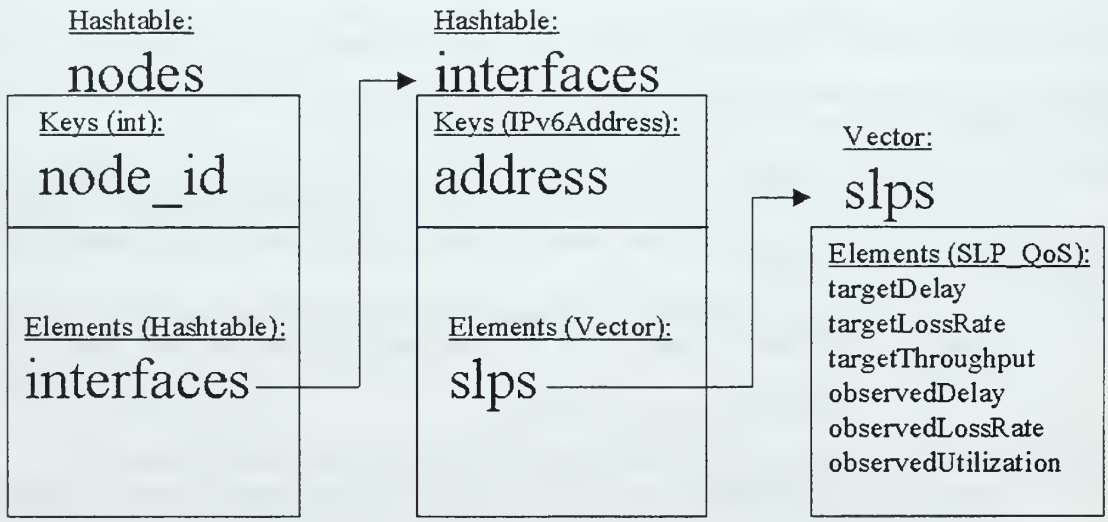


Figure 5.24. The `Nodes` Class Object. [From Ref. 36]

The `Paths` object is a hash table that uses the assigned `path id` of each router as the key (See Figure 5.25). The elements stored in this hash table are called `path objects`. A `path object` contains several attributes. Among these are a

hashtable maintaining information about flows that are assigned to this path. The *flow id* assigned to a flow is used as the key for the hashtable. The elements stored in the hashtable are *Flow_QoS* objects. A *Flow_QoS* object contains the negotiated and observed QoS parameters for the flow. Another attribute of the path object is a vector called *SLPSequence*. The *SLPSequence* vector stores an ordered list of the service level pipes that make up the path. These *ServiceLevelPipe* objects store the *IPv6 address* and service level of the service level pipe. [Ref. 36]

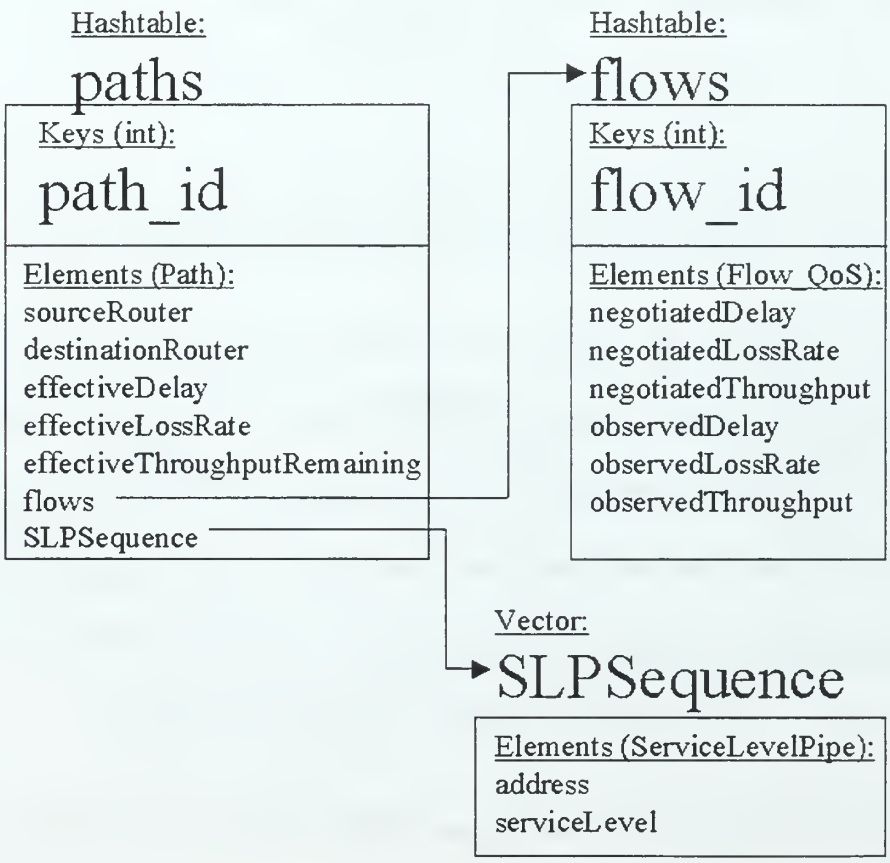


Figure 5.25. The *Paths* Class Object. [From Ref. 36]

The `links` object is a hash table that uses the derived `link id` of a network segment as the key (See Figure 5.26). The elements stored in this hash table are integers that describe the bandwidth of the link. [Ref. 36]

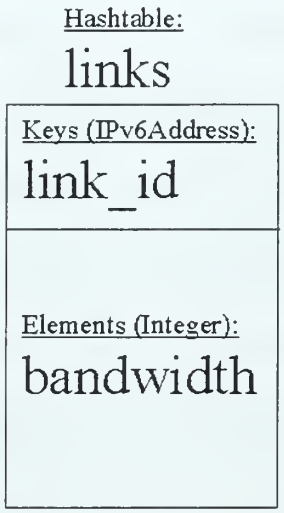


Figure 5.26. The `Links` Class Object. [From Ref. 36]

In order to determine the entire PIB size, formulas that give the sizes of main PIB elements (`nodes` hashtable, `paths` hashtable, and `links` hashtable) are derived and for each of them all sub elements' sizes are calculated. In Table 5.5, sizes of the elements forming the `nodes` hashtable are summarized. The size of the `nodes` hashtable, N_s, can be calculated by using the following equation:

$$N_s = NH_s + N_n\left(EH_s + \overline{ni}\cdot\left(EV_S + ns\cdot SLO_S\right)\right) \tag{5.6}$$

where N_n is the total number of nodes in PIB, $\overline{ni}$ the average number of interfaces residing on one router (typically $\overline{ni}$ is about three), and ns the number of supported service levels (right now four service levels are supported). Other notation is explained in Table 5.5.

Nodes		
Name	**Symbol**	**Size (bytes)**
Nodes Hastable (empty)	NH_s	416
Elements Hashtable (empty)	EH_s	35
Elements Vector (empty)	EV_s	71
Slp_QoS Object	SLO_s	29

Table 5.5. The `Nodes` Hastable Element Sizes.

In Table 5.6, sizes of the elements forming the `paths` hashtable are summarized. The size of the `paths` hashtable, P_s, can be calculated by using the following equation:

$$P_s = PH_s + N_p\left(PO_s + FO_s\left(\overline{nf} - 1\right) + SO_s\left(\overline{h} - 1\right)\right) \tag{5.7}$$

where N_p is the total number of paths in PIB, $\overline{nf}$ the average number of flows using a path, and $\overline{h}$ the average number of hops per path. Other notation is explained in Table 5.6.

Paths		
Name	**Symbol**	**Size (bytes)**
Paths Hastable (empty)	PH_s	176
Paths Object	PO_s	773
Flow_QoS Object	FO_s	40
ServiceLevelPipe Object	SO_s	41

Table 5.6. The `Paths` Hastable Element Sizes.

In Table 5.7, sizes of the elements forming the `links` hashtable are summarized. The size of the `links` hashtable, L_s, can be calculated by using the following equation:

$$L_s = LH_s + N_l \cdot R_s \tag{5.8}$$

where N_l is the total number of links in PIB. Other notation is explained in Table 5.7.

Links		
Name	**Symbol**	**Size (bytes)**
Links Hastable (empty)	LH_s	151
One record in Links Hashtable	R_s	47

Table 5.7. The `Links` Hastable Element Sizes.

In the typical SAAM region, first, if we assume that number of nodes in the PIB, N_n, is 70, then the size of the `nodes` hashtable, N_s, is calculated as 42,136 bytes by using Equation 5.6. Second, if we assume that total number of paths in the PIB, N_p, is 5000, average number of flows using a path, $\overline{nf}$, is 100, and average number of hops per path, $\overline{h}$, is 5, then the size of the paths hashtable, P_s, is calculated as 21,500,773 bytes by using the Equation 5.7. Third, if we assume that total number of links in the PIB, N_l, is 100, then the size of the links hashtable, L_s, is calculated as 621 bytes by using the Equation 5.8. Finally, when the sizes of the nodes hashtable, the paths hashtable, and the links hashtable are summed up, the size of a typical PIB is calculated as 21,543,530 bytes, which is approximately 20 MB.

b. Selected Approach

To provide the backup SAAM server with an up-to-date PIB, if the periodical replication of the original PIB from the primary SAAM server to the backup server approach is used, then the primary SAAM server will be required to send approximately 20 MB of data in every two or three seconds. This would introduce high

overhead on the network, especially on the routers that are directly connected to the servers.

On the other hand, when the duplicating and sending all router originated messages to both the primary SAAM server and the backup SAAM server approach is used, the overhead introduced to the network is negligible and traffic is spread out evenly. Additionally, since the backup SAAM server receives all the messages in parallel with the primary SAAM server, the backup server would always have an almost identical copy of the primary PIB. For these reasons, this approach is selected to provide the backup SAAM server with an up-to-date PIB.

5. Fault Treatment and Continued Service

The fault treatment and continued service phase is mainly focused on the states of the SAAM server after the failure has occurred. When the primary SAAM server fails and the backup SAAM server takes over the responsibility of the primary SAAM server, the system is no longer fault-tolerant. If the new primary SAAM server fails before the failed server is repaired and reconnected to the system, then the entire SAAM system will fail. Therefore, it is essential to repair and to reinstate the repaired server to the network as soon as possible. This phase can be performed in three subphases: *identification phase*, *repair phase*, and *reinstating phase*.

In the identification phase, the failed server is identified and is located. To speed up the identification phase, it is desirable to have an alert system that informs the administrator about the failed SAAM server. The alert can be in the form of displaying a message on the screen, a sound alarm, or an e-mail message to the administrator.

Once the failed SAAM server is identified, then it has to be repaired very quickly. In the repair phase, the component of the failed SAAM server containing the fault is identified. Then, the located faulty component is repaired. Due to the severity of the environmental disaster, repair might not always possible. In this case, a brand new server needs to be deployed. The repairing process is performed in the failed state of the SAAM server (See Figure 5.2). After repair, the SAAM server enters the repaired state and the reinstating phase starts.

In the reinstating phase, the administrator decides on reinstating the repaired SAAM server either as a backup to the current primary SAAM server or as a new primary. If the administrator wants the repaired SAAM server to serve as the backup server, then the repaired SAAM server enters the silent running state (See Figure 5.2). If the administrator wants to reinstate the repaired server as a primary server, then the repaired SAAM server enters the PIB synchronization state (See Figure 5.2) and builds its PIB from the LSA messages. Once the PIB is built, then the repaired server is ready to be the new primary SAAM server. Then, the repaired server sends the *primary server id* message to all the nodes in the region and enters the active running state. Any other active server, upon receiving the primary server id message, returns to the silent running state. Additionally, the routers, upon receiving the primary server id message will know about the new primary SAAM server.

B. INTEGRATION WITH THE EXISTING SOURCE SODE

In this section, the integration of the remote area fault tolerance implementation with the existing SAAM server source code is explained (see Appendices of Ref. 36 for the entire source code). The integration covers the failure detection, damage confinement and assessment, and failure recovery phases. Due to the time constraints, the fault treatment and continued service phases, and the handling of the simultaneous existence of two active servers are not implemented in this integration.

1. Packet Formats

In the current SAAM test environment all packets are encapsulated with an Ipv4 header to enable them to be passed on current Ipv4 networks. In the SAAM test environment two different types of packet formats are used; *demo packet* and *emulation packet*. The demo packets (see Figure 5.27) are used by the demo station* to initialize and test the routers and the servers of a SAAM region. Specifically, the demo packet contains a SAAM packet encapsulated within an Ipv4 packet.

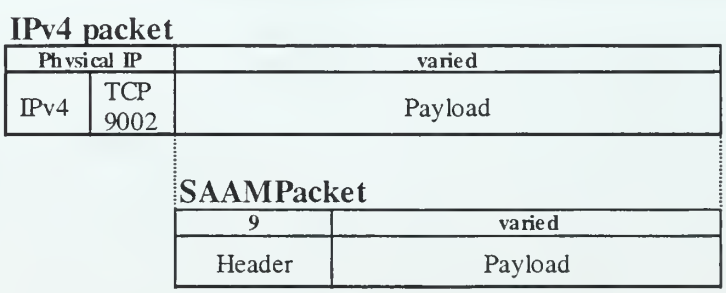


Figure 5.27. Demo Packet Structure.

* Demo station is a Java application used by the administrator to build and test a SAAM network remotely.

On the other hand, the emulation packet (see Figure 5.28) format is used between the components of the actual SAAM architecture. Emulation packets are structured to reflect the layers of the *emulated* protocol stack (for a description of the emulated stack, see Ref. 36). Each emulation packet contains an outer IPv4 header followed by a TCP header for port 9002 to enable the packet to travel on existing networks. The MAC field corresponds to the link layer of the *emulated* protocol stack. The *emulated* NIC strips off this field. The remaining portion of the IPv4 payload consists of an IPv6 packet, which is the true protocol data unit for the SAAM network.

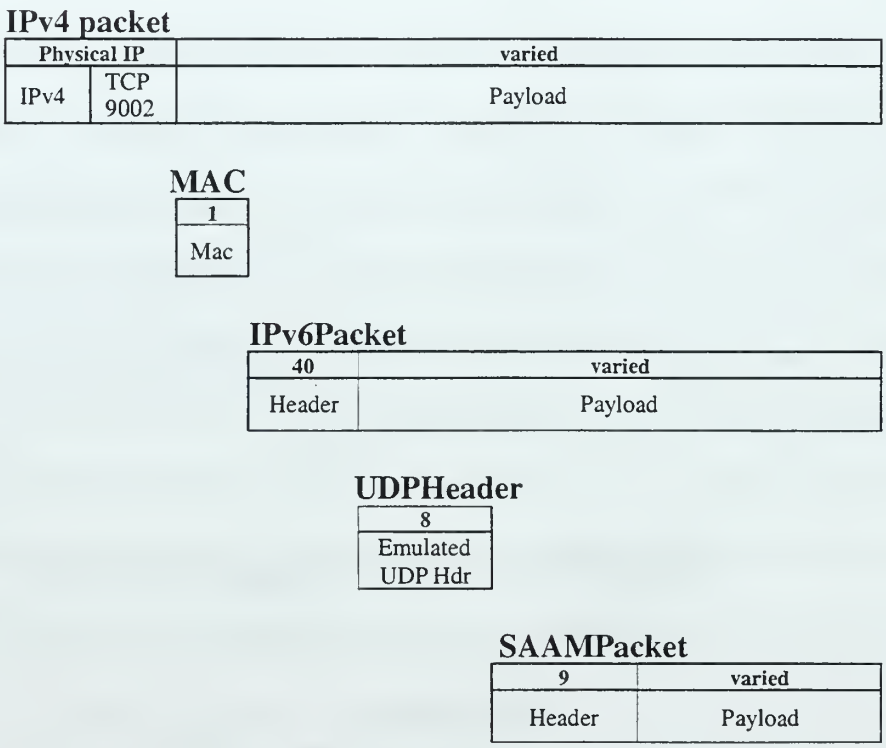


Figure 5.28. Emulation Packet Structure.

The nodes in a SAAM network need to pass SAAM protocol specific messages to each other. These messages are called SAAM packets and consist of a header and a payload (see Figure 5.29). Each SAAM packet is encapsulated in an emulation packet (see Figure 5.28). A SAAM packet may contain several SAAM control messages. A server may send several flow routing table entries to a router at once by encapsulating those entries within a single SAAM packet. The header of a SAAM packet contains an eight-byte *time-stamp* and a one-byte *number of messages contained in the payload.* The payload portion could contain either messages or resident agents.

The `HeartbeatQuery` and the `HeartbeatResponse` messages exchanged between the two servers are transferred in the payload portion of a SAAM packet (see Figure 5.30). The `HeartbeatQuery` message contains a one-byte *type* field (the type value of the `HeartbeatQuery` message is two) and a two-byte *sequence number* field. In comparison, the `HeartbeatResponse` message contains a one-byte *type* field (type number of the `HeartbeatResponse` message is three), a two-byte *sequence number* field, and a four-byte *last used flow ID* field.

The *last used flow ID* field of the `HeartbeatResponse` message informs the backup server about the flow ID value assigned by the primary server. The *last used flow ID* filed is required by the backup server because the backup server is not aware of the currently allocated flow IDs during its silent running state. However, when the backup server takes over the job of the primary server, it needs to assign the new flows to flow IDs that are not currently assigned. The last used flow ID is a three-byte number and whenever a new flow ID is assigned, it is incremented by one. Therefore, after it takes

over the primary server's job, the backup server can safely assign new flow IDs starting from the last used flow ID plus one.

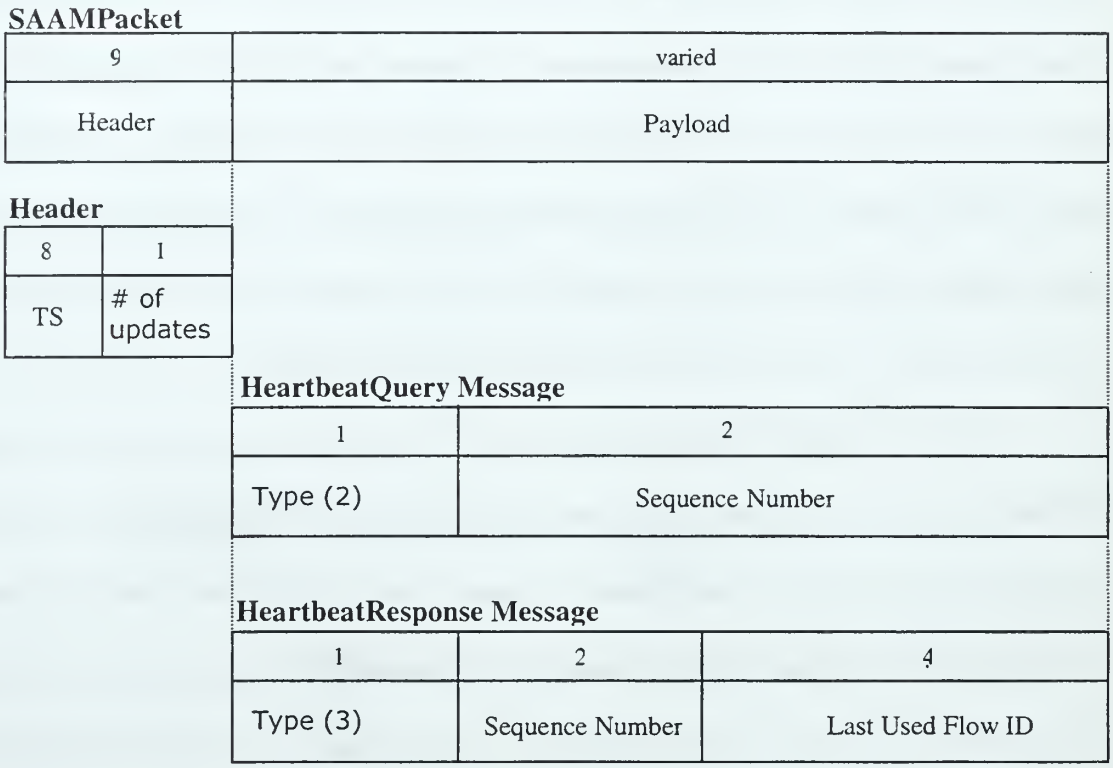


Figure 5.29. Heartbeat Message Structures.

2. Integration of Error Detection Mechanism

For the integration of the designed accelerated heartbeat protocol prototype, four new Java class files were added to the existing SAAM server source code. In the following sections, the specifications and the purposes of these Java classes are explained in detail.

a. `HeartbeatQuery` *Class*

The `HeartbeatQuery` class (see Appendix C) is extended from the `Message` class (see Appendix D). The `HeartbeatQuery` class is used by the backup SAAM server as a heartbeat query message. This class file encapsulates the sequence number field shown in Figure 5.29. The first type field of the heartbeat query message is implemented as a data member of the `Message` class. The `HeartbeatQuery` class has three private data members; `bytes`, `sequenceNumber`, and `counter`. The `bytes` is a Java byte array object that stores the byte code representation of this class. The `sequenceNumber` is a `short` (primitive data type) that is used for numbering the heartbeat query messages. The `counter` is a `static short` (primitive data type) used for assigning a value to the `sequenceNumber`. Whenever a new instance of the `HeartbeatQuery` class is created, the `counter` is incremented by one and its value is assigned to `sequenceNumber`.

The `HeartbeatQuery` class has two constructors. The first constructor is a parameterless constructor. This constructor is used by the backup server to create an instance of this class to send to the primary server. The second constructor accepts a byte array (a byte code representation of this class) as a parameter. This constructor is used by the primary server to reconstruct the message instance by using the byte array received from the backup server. The `HeartbeatQuery` class also has two `get` methods that return the data members of `sequenceNumber` and `bytes`. Additionally, it has `length` and `toString` methods that return the length of the byte array and the string representation of this class, respectively.

b. `HeartbeatResponse` *Class*

The `HeartbeatResponse` class (see Appendix C) is also extended from the `Message` class. The `HeartbeatResponse` class is used by the primary SAAM server as a heartbeat response message. This class file encapsulates the sequence number and the last used flow ID fields shown in Figure 5.29. The `HeartbeatResponse` class has three private data members: `bytes`, `sequenceNumber`, and `lastUsedFlowID`. The `bytes` is a Java byte array object that stores the byte code representation of the `HeartbeatResponse` class. The `sequenceNumber` is a `short` (primitive data type) used for numbering the heartbeat response messages. The initialization of the `sequenceNumber` in this class is different than the `HeartbeatQuery` class's `sequenceNumber` data member initialization. Instead of using a counter value, the sequence number of the last received heartbeat query message is used.

The `HeartbeatResponse` class also has two constructors. The first constructor receives the `sequenceNumber` (`short`) and the `lastUsedFlowID` (`int`) as parameters, and initializes the data members of this class. This constructor is used by the primary server to create an instance of this class to send to the backup server. The second constructor receives a byte array (a byte code representation of this class) as a parameter. This constructor is used by the backup server to reconstruct the message instance by using the byte array representation received from the primary server. The `HeartbeatResponse` class also has three `get` methods that return the data members of this class (`bytes`, `lastUsedFlowID`, and `sequenceNumber`). This class also

has `length` and `toString` methods that return the length of the byte array and the string representation of this class, respectively. Additionally, the HeartbeatResponse class has two `set` methods that are used for setting the values of the `sequenceNumber` and the `lastUsedFlowID`.

c. `HeartbeatController` Class

The `HeartbeatController` class (see Appendix C) is created by the `Server` class. The `HeartbeatController` class is responsible for periodically sending the heartbeat query messages and performing a check on the time constraints of the heartbeat response messages.

In order to periodically send the heartbeat query messages, the `HeartbeatController` class uses a `Timer` called `querySendTimer`. The initial delay of the `querySendTimer` is set to t_{max}. The `querySendTimer` is a repeating timer, and is restarted with its initial delay every time when it expires. Whenever the `querySendTimer` expires, the `actionPerformed` method of the `QuerySendTimerHandler` class (inner class of the `HeartbeatController` class) is executed. From this `actionPerformed` method, the `sendHertbeatQuery` method of the `Server` class is called.

In order to perform a check on the time constraints of the heartbeat response messages, the `HeartbeatController` class uses another `Timer` called `responseControlTimer`. The `responseControlTimer`'s initial delay is set to the half of the t_{max} value. Whenever a correct heartbeat response message is received

from the primary server, the *responseControlTimer* is stopped, and whenever a new heartbeat query message is sent it is restarted with the half of the current interval time. Therefore, as long as the heartbeat response messages are received properly the *responseControlTimer* never expires. If the backup server sends a heartbeat query message, but does not receive a heartbeat response message within the first half of the current period, then the *responseControlTimer* expires. Whenever the *responseControlTimer* expires, the *actionPerformed* method of the *responseControlTimerHandler* class, (an inner class of the *HeartbeatController* class), is executed.

In this *actionPerformed* method, the *sendHertbeatQuery* method of the *Server* class is called, and the length of the next interval period is reduced by half. If the length of the next interval period ever becomes less than the t_{min}, then the time of the last received DCM message is checked. If a new DCM is received within the last t_{max} period, then a failure declaration is not decleared. If a new DCM is not received within the last t_{max} period, then failure declaration of the primary server is declared by calling the *setIsMainDown* method of the *Server* class.

d. BannerFrame Class

The *BannerFrame* class (see Appendix C) is a GUI component added to the current GUI of the *Server* class to display some accelerated heartbeat protocol information to the user. Specifically, the *BanerFrame* class is a Java class extended from the *JFrame*. It has only one constructor that accepts a string for display.

Additionally, the `BannerFrame` class has three `set` methods. The `setFrameText` method is used to change the currently displayed text. The `setBackgroundColor` method is used to change the background color of the frame, and the `setForegroundColor` is used to change the color of the displayed text on the frame.

Initially, according to the server type either "THIS IS THE PRIMARY SERVER" text (see Figure 5.32) or "THIS IS THE BACKUP SERVER" text (see Figure 5.31) is displayed on the `bannerFrame`. If the backup server fails to receive a heartbeat response message, then "PRIMARY SERVER MISSED *n* RESPONSE" text is displayed (*n* is the number of missed messages). Additionally, if the backup server declares the failure of the primary server, then "PRIMARY SERVER IS DOWN" (see Figure 5.33) text is displayed.

3. Modifications Done on the Existing Source Code

In order to integrate the remote area fault tolerance implementation with the existing source code, some of the existing class files were modified. (The source codes of all modified files can be found in Appendix D). In the following sections, the modifications made to these class files are explained.

a. `Message` Class

In the previous implementation, SAAM Packet supported only two types of payloads, *type-0* and *type-1*. The type-0 was used for the resident agents and type-1 was used for messages. Previously, messages were identified by using their Java class names. In the SAAM packet payload, in addition to the message bytecode, the length of

the message bytecode, the message's Java class name, and the length of the class name were required fields. Due to these required fields for each message, the size of the payload was relatively very large.

In our design implementation, all messages are given different type numbers and identified by using these type numbers. For example, the heartbeat query message and the heartbeat response message are given type numbers two and three, respectively. When the type numbers are used to identify the messages, length of the message bytecode, the message's Java class name, and the length of the class name fields are no longer required. Therefore, they are removed from the structure of the message in the SAAM packet.

b. `Server` Class

In order to support the implementation of remote area fault tolerance, the following methods are added to the `Server` class:

- `void initHeartbeat()`: This method is invoked from the `processConfigutation ()` method when a configuration message is received from the demo station. First, `initHeartbeat()` method properly initializes the `BannerFrame` according to the server type received from the configuration message. Second, if the server type is "backup," then this method creates a HeartbeatController class object, and also initiates the heartbeat query message sending process.
- `void processHeartbeatQuery(HeartbeatQuery hbq)`: This method is invoked when a heartbeat query message is received from the

backup SAAM server. This method immediately sends the heartbeat response message upon receiving the heartbeat query message.

- *void processHeartbeatResponse(HeartbeatResponse hbr)*: This method is invoked when a heartbeat response message is received from the primary SAAM server. In this method, the sequence number of the received heartbeat response message is controlled. If the sequence number is the expected one or it exists in the *recentMissedSequences* vector, then the heartbeat response message is accepted, otherwise it is ignored. If the received heartbeat response message is accepted, then the *responseControlTmer* in the *HeartbeatControll* class is stopped.
- *void sendHeartbeatQuery()*: This method is invoked when a *querySendTimer* of the *HeartbeatController* class expires. It sends a heartbeat query message to the primary SAAM server.
- *void setMainDown()*: This method is invoked from the *HeartbeatController* class when the failure of the primary server is declared. It sets the *isMainDown* Boolean data member to "true."
- *void addRecentMissedSequences()*: This method adds the sequence number of the missed heartbeat response message to the *recentMissedSequences* vector.
- *void clearRecentMissedSequences()*: This method deletes all of the elements in the *recentMissedSequences* vector.

- *void printRecentMisses()*: This method displays the elements of the *recentMissedSequences* vector on the server GUI.
- *long getLastResponseTime()*: This method returns the *lastResponseTime* data member.
- *void display(String str)* : This method displays the passed string on the server GUI.

c. *ServerAgent Class*

The *ServerAgent* class installs itself as a resident agent and registers with the *ControlExecutive* to process specific message types. Previously, the *ServerAgent* were registered with the *ControlExecutive* for only *Hello*, *FlowRequest*, and *LinkStateAdvertisement* messages. Additionally, it was not capable of handling the heartbeat query and heartbeat response messages.

In order to make the *ServerAgent* to receive the heartbeat query and heartbeat response messages, first it is also registered with the *ControlExecutive* for the heartbeat query and the heartbeat response messages. Second, in order to make the *ServerAgent* call appropriate methods of the *Server* class, two new *if* statements are added to *the processMessage()* method for each heartbeat message type.

d. *PacketFactory Class*

The *PacketFactory* class is used to build SAAM packets for sending or receiving SAAM packets and extracting their atomic elements. In the *PacketFactory* class, messages are built by the *append()* method and received by

the *processPacket()* method. Previously, both the *append()* and the *processPacket()* methods were not capable of handling the heartbeat query and the heartbeat response messages. Therefore, these two methods were modified. Specifically, in each method, two new cases for the heartbeat query and the heartbeat response messages were added to the existing *switch* statements.

C. TESTING

This section describes the test environment that is used to test the integration of the remote area fault tolerance implementation with the existing SAAM source code. Currently, the SAAM components (routers and servers) are emulated to operate on the existing IPv4 networks. Because of the emulation overhead, the system time has to be scaled on a per node basis. Therefore, it is not possible to accurately test time related constraints of the fault tolerance design. Instead, the focus is on testing the functionality of the remote area fault tolerance implementation.

1. Testbed

To test the functionality of the implemented remote area fault tolerance features, the test bed shown in Figure 5.30 was developed. The test topology consisted of five emulated SAAM routers, two emulated SAAM servers (one primary and one backup), and one demo station.

The demo station, located on "kati3" whose IPv4 address 131.120.8.79, is hard-coded with all of the information (see Table 5.8) necessary to initialize the server and all routers (see Appendix D for the source file). During the initialization of the nodes in the testbed, the demo station uses the demo packets shown in Figure 5.27.

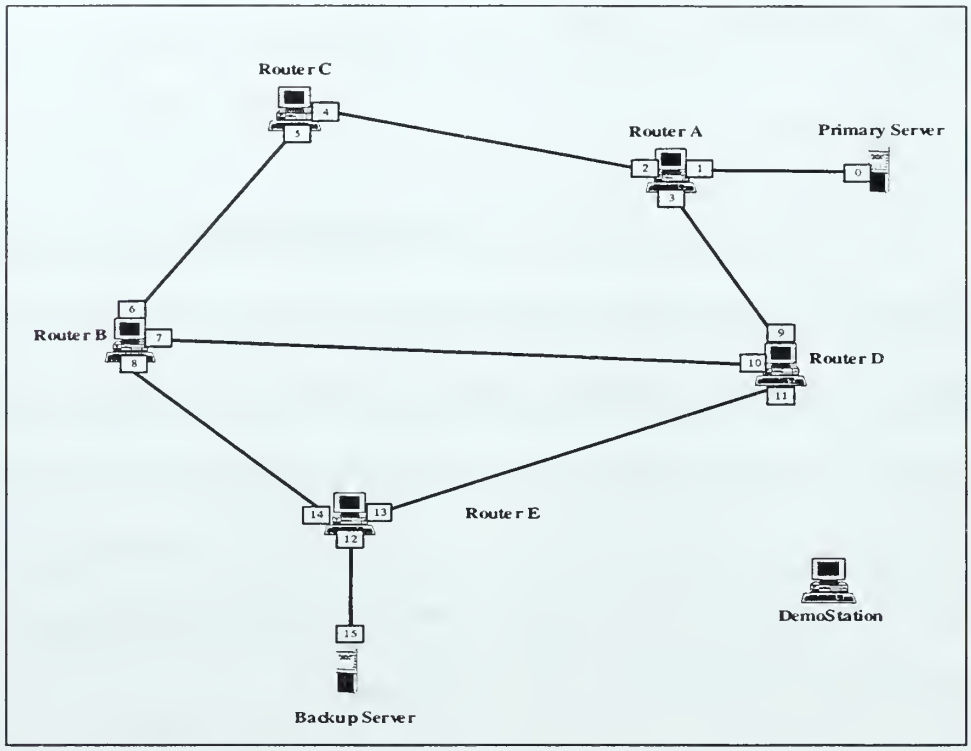


Figure 5.30. Test Topology.

NODE NAME	NODE IPv6 ADDR.	NODE IPv4 ADDR.	EMULATED MAC ADDR.	INTERFACE IPv6 ADDR.
Primary Server	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	131.120.8.135	0	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
Backup Server	99.99.99.99.8.0.0.0.0.0.0.0.0.0.0.2	131.120.8.132	15	99.99.99.99.8.0.0.0.0.0.0.0.0.0.0.2
Router A	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1	131.120.8.134	1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
			2	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1
			3	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1
Router B	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	131.120.8.139	6	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2
			7	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1
			8	99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1
Router C	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	131.120.9.76	4	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2
			5	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1
Router D	99.99.99.99.7.0.0.0.0.0.0.0.0.0.0.1	131.120.9.73	9	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2
			10	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2
			11	99.99.99.99.7.0.0.0.0.0.0.0.0.0.0.1
Router E	99.99.99.99.8.0.0.0.0.0.0.0.0.0.0.1	131.120.9.71	12	99.99.99.99.8.0.0.0.0.0.0.0.0.0.0.1
			13	99.99.99.99.7.0.0.0.0.0.0.0.0.0.0.2
			14	99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2

Table 5.8. Specifications of the Nodes in the Test Topology.

Additionally, during the initialization of the nodes, the demo station sends a special message, called a *Configuration* message, to both servers. The configuration message is used to differentiate the primary and the backup server. Actually, the configuration message is used for the initialization of the auto-configuration metrics, and is not part of the remote area fault tolerance implementation. Therefore, the details of the message content will not be explained here. However, the server nodes use the *Server_Type* (one byte) field of the configuration message to determine to behave either as a primary or as a backup. Specifically, if the *Server_Type* field is zero, then the server node is initialized as a primary server. On the other hand, if the *Server_Type* field is one, then the server node is initialized as a backup server.

2. Tests Performed

In order to evaluate the functionality of the remote area fault tolerance implementation, five different cases were tested. In the first test case, the failure detection capability of the implementation was tested. In the second test case, the effect of some heartbeat response message losses on the failure detection was tested. In the third test case, the functionality of the message-numbering scheme was tested. In the fourth test case, the usage of the DCM messages as unsolicited heartbeat messages was tested. Finally, in the fifth test case, the functionality of the provided $t_{\max} \geq \frac{8C}{7}$ relationship between the $t_{\max}$ (time interval between regular heartbeat query messages) and C (time interval between the DCM messages) was tested.

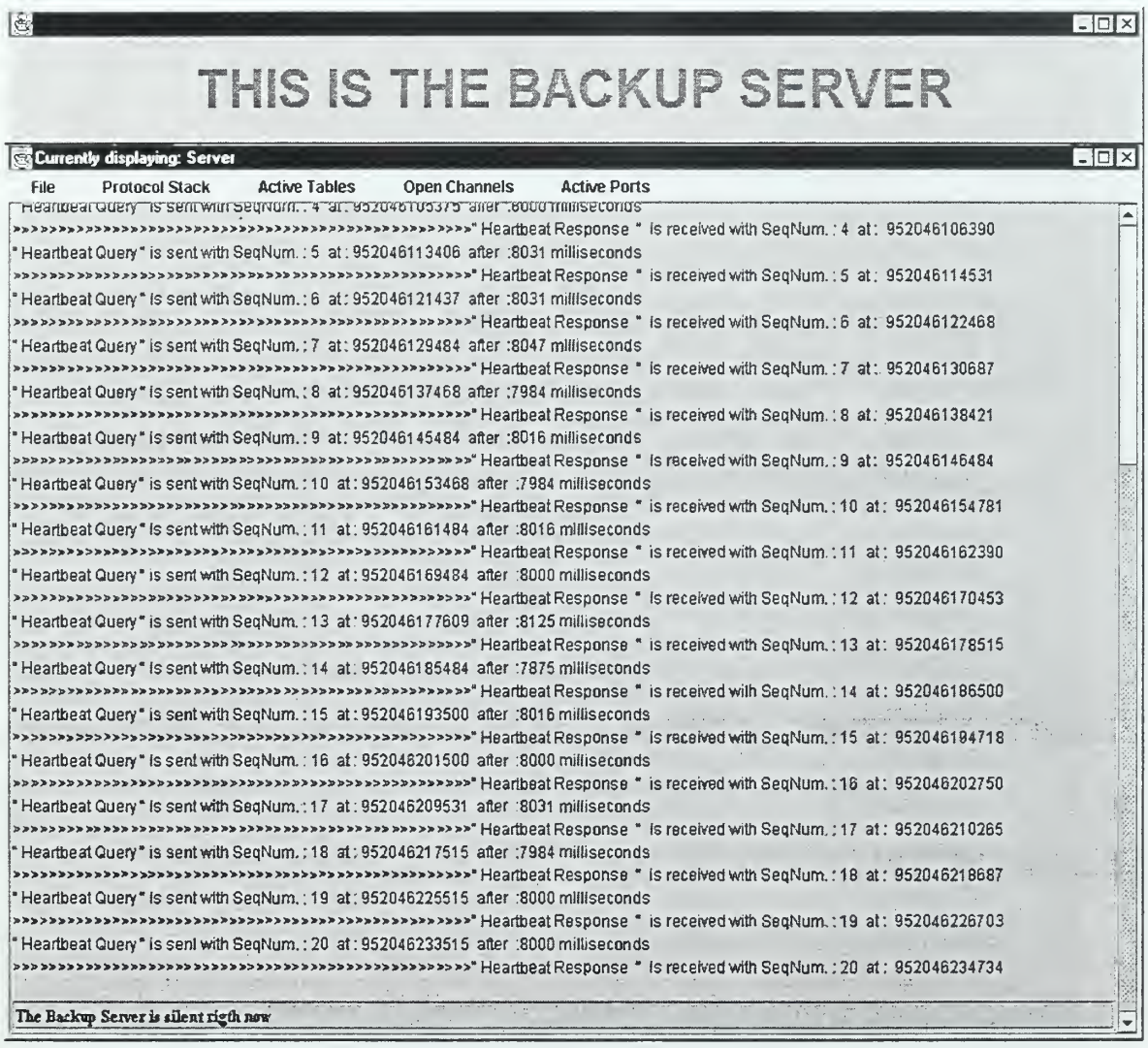
In the first four tests, the desired test conditions were obtained by adding an additional "fault injecting code" to the `Server` class file. Specifically, a different boolean data member was provided (`testCase1`, `testCase2`, `testCase3` and `testCase4`), and for each test case only one of these boolean values was set to "true" (see Appendix D). These boolean values were used to activate or deactivate the code sections that created the test conditions to happen. In the fifth test, the desired test condition was obtained by manually shutting down one the routers that was the part of the control channel. In the following sections, tests performed and their results are explained in detail.

a. Failure Detection Test

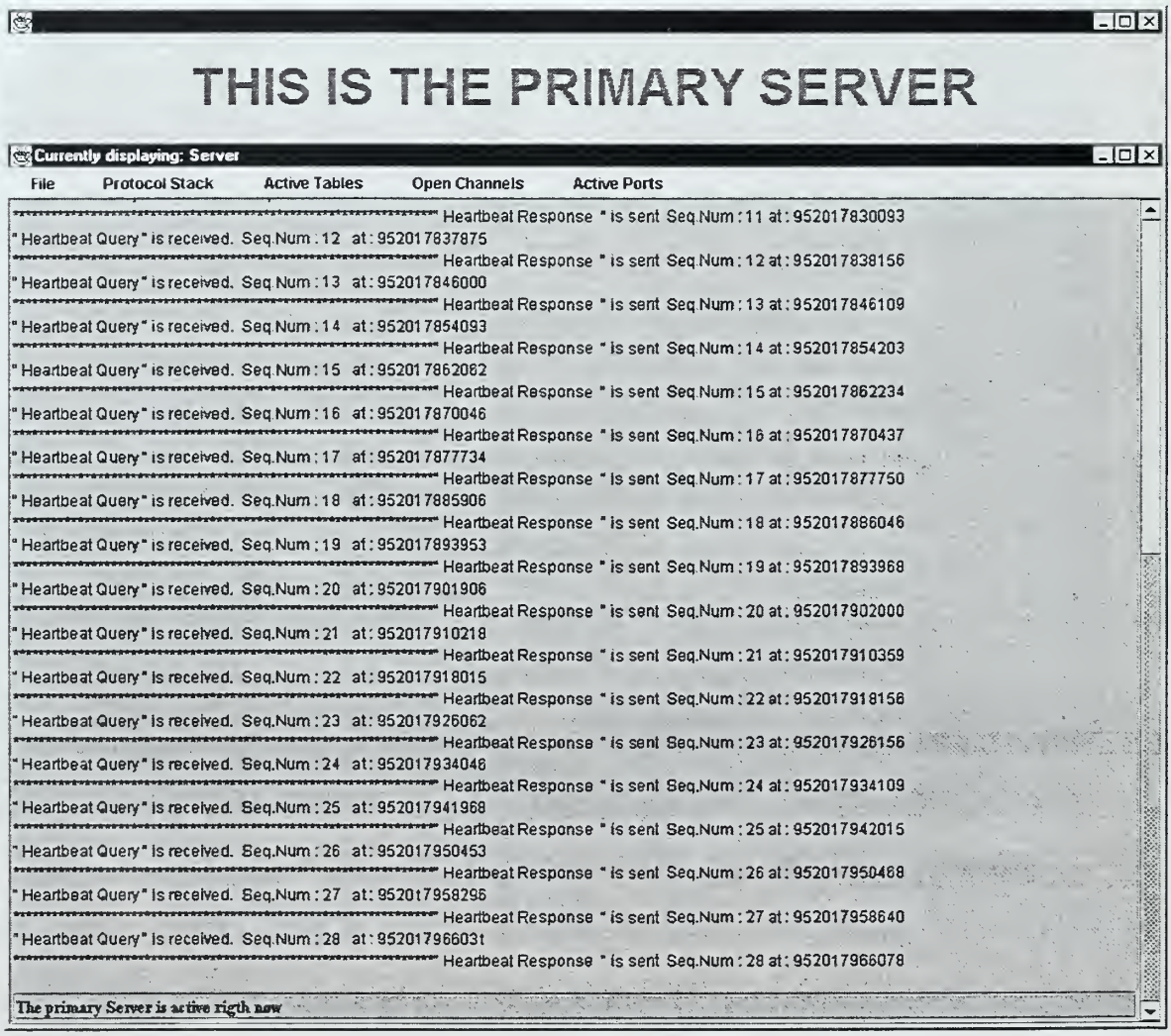
The failure detection test (`testCase1`) was performed in order to test the basic failure detection capability of the remote area fault tolerance implementation. After the initialization of the test topology (shown in Figure 5.30), control channels were established by the auto-configuration mechanism. Then, the backup server started to monitor the health of the primary server by periodically sending heartbeat query messages. In this test, the values of the t_{max} and t_{min} were set to 8.0 seconds and 0.6 seconds, respectively.

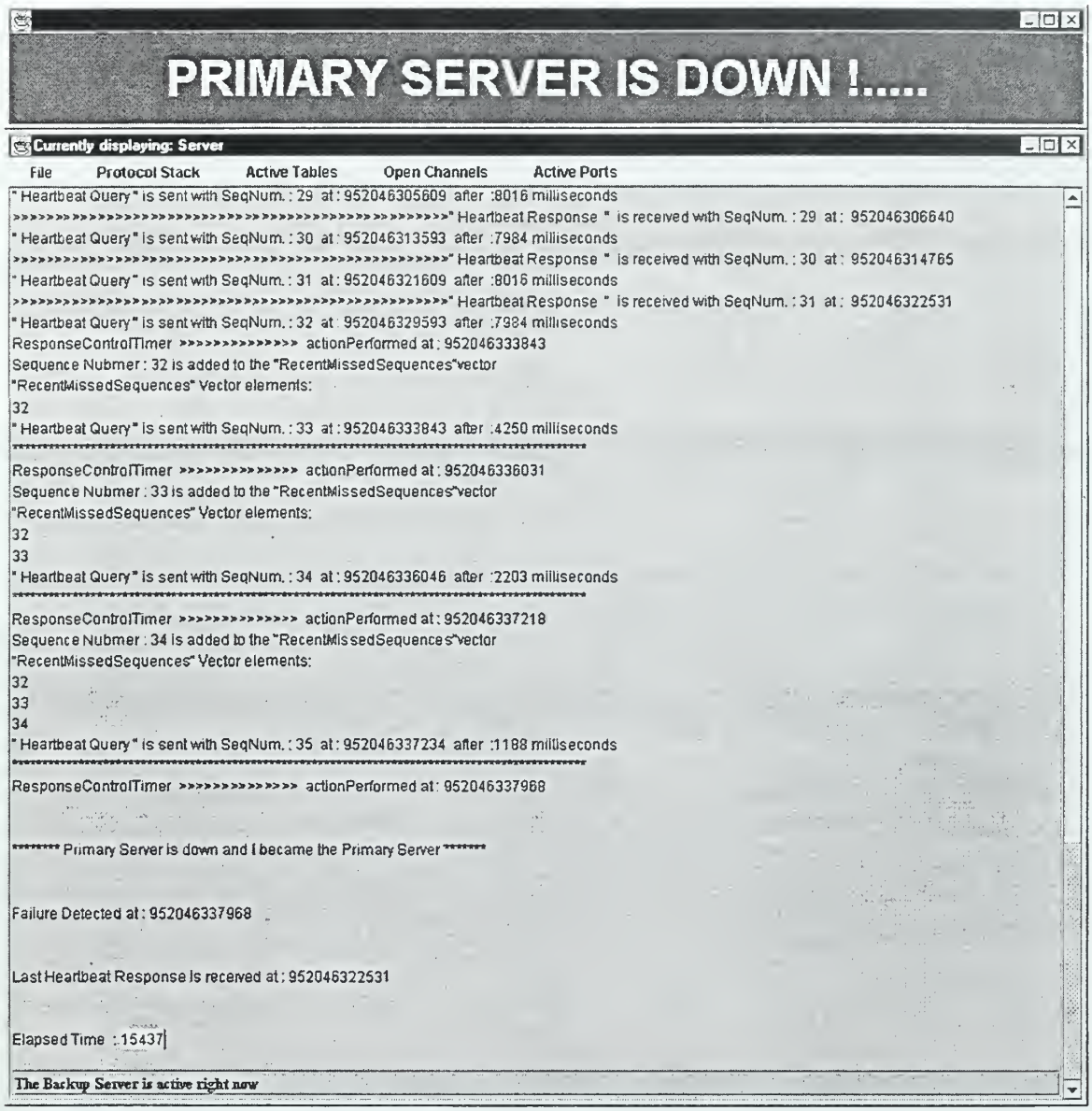
During the test, we observed that the heartbeat query and the heartbeat response messages were exchanged properly between two servers. Specifically, the backup server was sending the heartbeat query messages every eight seconds as specified with t_{max} value (see Figure 5.31). Additionally, the primary server was immediately

sending the heartbeat response messages upon receiving the heartbeat query messages (see Figure 5.32).


THIS IS THE BACKUP SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
>>>

failure detection delay was 15473 milliseconds and calculated by subtracting the last received heartbeat response message time from the failure detection time. During this failure detection period, the backup server queried the primary server's heath four times. As a result, both the primary and the backup servers behaved according to the accelerated heartbeat protocol specifications. Consequently, the implementation successfully passed the failure detection test.


THIS IS THE PRIMARY SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
********** Heartbeat Response " is sent Seq.Num : 11 at : 952017830093
" Heartbeat Query " is received. Seq.Num : 12 at : 952017837875
********** Heartbeat Response " is sent Seq.Num : 12 at : 952017838156
" Heartbeat Query " is received. Seq.Num : 13 at : 952017846000
********** Heartbeat Response " is sent Seq.Num : 13 at : 952017846109
" Heartbeat Query " is received. Seq.Num : 14 at : 952017854093
********** Heartbeat Response " is sent Seq.Num : 14 at : 952017854203
" Heartbeat Query " is received. Seq.Num : 15 at : 952017862062
********** Heartbeat Response " is sent Seq.Num : 15 at : 952017862234
" Heartbeat Query " is received. Seq.Num : 16 at : 952017870046
********** Heartbeat Response " is sent Seq.Num : 16 at : 952017870437
" Heartbeat Query " is received. Seq.Num : 17 at : 952017877734
********** Heartbeat Response " is sent Seq.Num : 17 at : 952017877750
" Heartbeat Query " is received. Seq.Num : 18 at : 952017885906
********** Heartbeat Response " is sent Seq.Num : 18 at : 952017886046
" Heartbeat Query " is received. Seq.Num : 19 at : 952017893953
********** Heartbeat Response " is sent Seq.Num : 19 at : 952017893968
" Heartbeat Query " is received. Seq.Num : 20 at : 952017901906
********** Heartbeat Response " is sent Seq.Num : 20 at : 952017902000
" Heartbeat Query " is received. Seq.Num : 21 at : 952017910218
********** Heartbeat Response " is sent Seq.Num : 21 at : 952017910359
" Heartbeat Query " is received. Seq.Num : 22 at : 952017918015
********** Heartbeat Response " is sent Seq.Num : 22 at : 952017918156
" Heartbeat Query " is received. Seq.Num : 23 at : 952017926062
********** Heartbeat Response " is sent Seq.Num : 23 at : 952017926156
" Heartbeat Query " is received. Seq.Num : 24 at : 952017934046
********** Heartbeat Response " is sent Seq.Num : 24 at : 952017934109
" Heartbeat Query " is received. Seq.Num : 25 at : 952017941968
********** Heartbeat Response " is sent Seq.Num : 25 at : 952017942015
" Heartbeat Query " is received. Seq.Num : 26 at : 952017950453
********** Heartbeat Response " is sent Seq.Num : 26 at : 952017950468
" Heartbeat Query " is received. Seq.Num : 27 at : 952017958296
********** Heartbeat Response " is sent Seq.Num : 27 at : 952017958640
" Heartbeat Query " is received. Seq.Num : 28 at : 952017966031
********** Heartbeat Response " is sent Seq.Num : 28 at : 952017966078
The primary Server is active rigth now
Figure 5.32. The Primary SAAM Server GUI During the Failure Detection Test.


PRIMARY SERVER IS DOWN !.....
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query " is sent with SeqNum. : 29 at : 952046305609 after :8016 milliseconds
>>>

server and receives a heartbeat response message within the first half of the current period, then the backup server makes the length of the next period t_{max} (irrespective of the length of the current period). Therefore, if the t_{max} and the t_{min} values are assigned allowing n heartbeat response message misses, then less than n consecutive heartbeat response message misses should not result in failure detection.

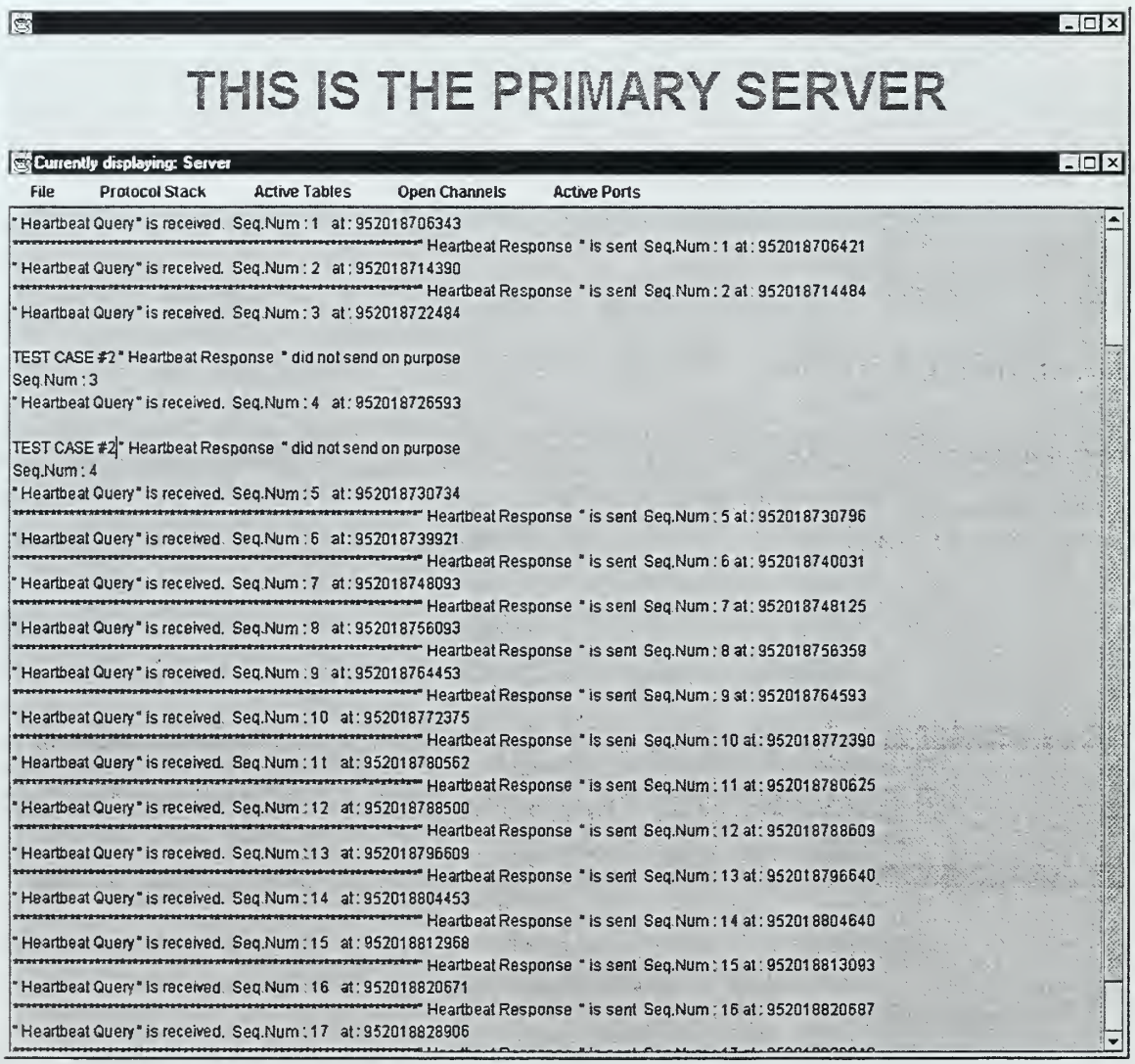
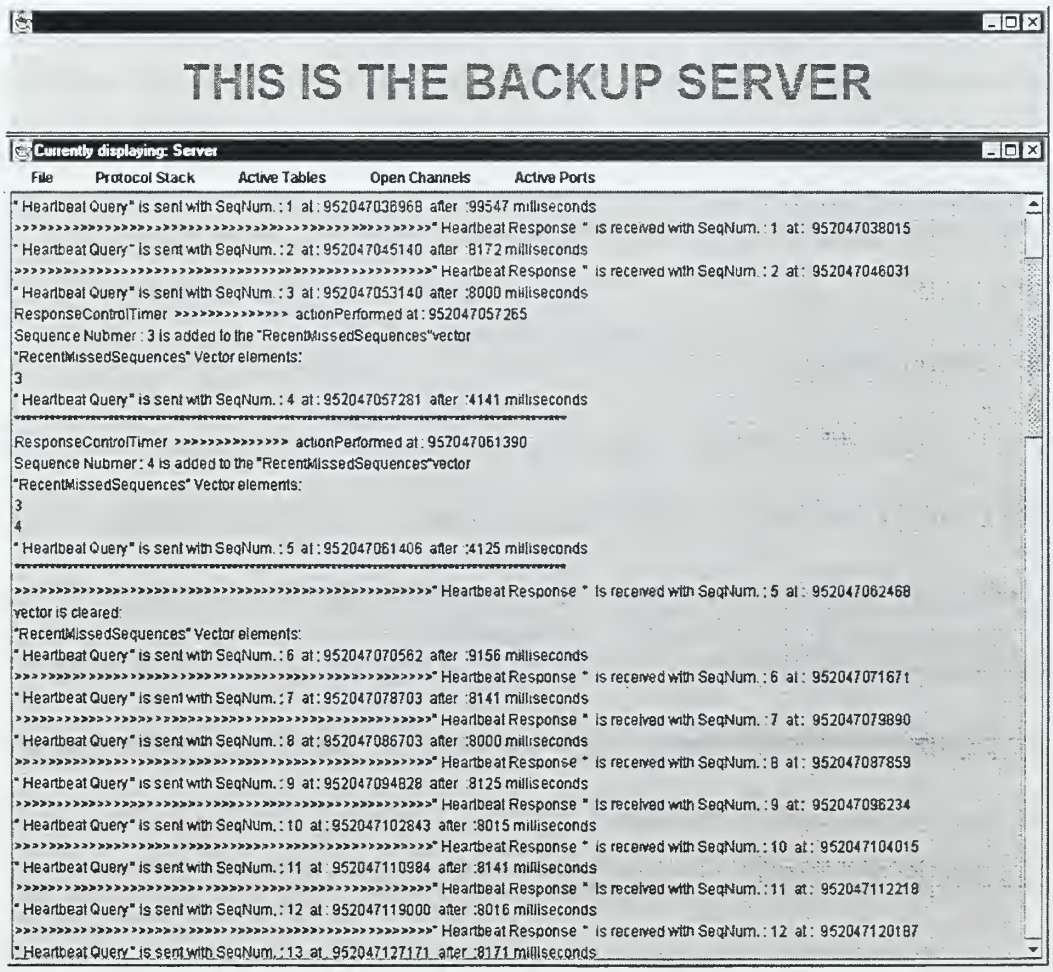

THIS IS THE PRIMARY SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query " is received. Seq.Num : 1 at : 952018706343
" Heartbeat Response " is sent Seq.Num : 1 at : 952018706421
" Heartbeat Query " is received. Seq.Num : 2 at : 952018714390
" Heartbeat Response " is sent Seq.Num : 2 at : 952018714484
" Heartbeat Query " is received. Seq.Num : 3 at : 952018722484
TEST CASE #2 " Heartbeat Response " did not send on purpose
Seq.Num : 3
" Heartbeat Query " is received. Seq.Num : 4 at : 952018726593
TEST CASE #2 " Heartbeat Response " did not send on purpose
Seq.Num : 4
" Heartbeat Query " is received. Seq.Num : 5 at : 952018730734
" Heartbeat Response " is sent Seq.Num : 5 at : 952018730796
" Heartbeat Query " is received. Seq.Num : 6 at : 952018739921
" Heartbeat Response " is sent Seq.Num : 6 at : 952018740031
" Heartbeat Query " is received. Seq.Num : 7 at : 952018748093
" Heartbeat Response " is sent Seq.Num : 7 at : 952018748125
" Heartbeat Query " is received. Seq.Num : 8 at : 952018756093
" Heartbeat Response " is sent Seq.Num : 8 at : 952018756359
" Heartbeat Query " is received. Seq.Num : 9 at : 952018764453
" Heartbeat Response " is sent Seq.Num : 9 at : 952018764593
" Heartbeat Query " is received. Seq.Num : 10 at : 952018772375
" Heartbeat Response " is sent Seq.Num : 10 at : 952018772390
" Heartbeat Query " is received. Seq.Num : 11 at : 952018780562
" Heartbeat Response " is sent Seq.Num : 11 at : 952018780625
" Heartbeat Query " is received. Seq.Num : 12 at : 952018788500
" Heartbeat Response " is sent Seq.Num : 12 at : 952018788609
" Heartbeat Query " is received. Seq.Num : 13 at : 952018796609
" Heartbeat Response " is sent Seq.Num : 13 at : 952018796640
" Heartbeat Query " is received. Seq.Num : 14 at : 952018804453
" Heartbeat Response " is sent Seq.Num : 14 at : 952018804640
" Heartbeat Query " is received. Seq.Num : 15 at : 952018812968
" Heartbeat Response " is sent Seq.Num : 15 at : 952018813093
" Heartbeat Query " is received. Seq.Num : 16 at : 952018820671
" Heartbeat Response " is sent Seq.Num : 16 at : 952018820687
" Heartbeat Query " is received. Seq.Num : 17 at : 952018828906

Figure 5.34. The Primary Server GUI in Heartbeat Response Message Loss Test.

In this test, the values of the t_{max} and t_{min} were 8.0 seconds and 0.6 seconds, respectively. Consequently, only four heartbeat response message misses were allowed. To perform the test, only the `testCase2` was set to "true" among the test case booleans of the `Server` class. In the activated code portion, the primary server performed a check on the sequence numbers of the heartbeat query messages. In order to simulate the heartbeat response message losses, the primary server intentionally did not send the heartbeat response messages to the heartbeat query messages with the sequence numbers three and four (see Figure 5.34).


THIS IS THE BACKUP SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query" is sent with SeqNum. : 1 at : 952047036968 after :99547 milliseconds
>>>

The backup server did not receive a heartbeat response message for heartbeat query message number three (see Figure 5.35). Then, the backup server reduced the interval value to four seconds and sent heartbeat query message number four. However, the backup server did not receive a heartbeat response message for the heartbeat query message number four either. Then, it reduced the interval value to two seconds and sent the heartbeat query message number five. This time, the backup server received the heartbeat response message and made the length of the next period again eight seconds, which was the t_{max} value.

During the test we observed that two consecutive heartbeat response message misses did not result in false failure detection. Additionally, the backup server operated as specified in the rules of the accelerated heartbeat protocol. Consequently, the implementation successfully passed the heartbeat response message loss test.

c. Message Numbering Scheme Test

The message numbering scheme test (`testCase3`) was performed in order to test the functionality of the sequence numbers (see Section A.2.f) used in heartbeat messages. In order to perform a check on the message sequence numbers, the backup server utilizes a Java Vector called `recentMissedSequences`. If the backup server fails to receive a heartbeat response message, then it stores the sequence number of the last heartbeat query message's sequence number in the vector. Then, whenever the backup server receives a heartbeat response message with a different than the expected one, the backup server searches through the vector to find the sequence number of the received heartbeat response message. If the backup server finds the sequence number in

the vector, then it accepts the heartbeat response message and clears the vector. Otherwise, it ignores the heartbeat response message. Additionally, if the backup server receives a heartbeat message with the correct sequence number, and if the vector is not empty, then the backup server clears the vector.

In this test, the values of the t_{max} and t_{min} were 8.0 seconds and 0.6 seconds, respectively. To perform the test, among the test case booleans of the `Server` class only the `testCase3` was set to "true". The test was performed in two steps. In the first step, the primary server did not respond to the heartbeat query message with the sequence number of 11 (see Figure 5.36). However, the primary server responded to the 12th heartbeat query message with the heartbeat response message that had the sequence number of 11. The backup server found the sequence number three in the `recentMissedSequences` vector, and accepted the message and cleared the vector (see Figure 5.37).

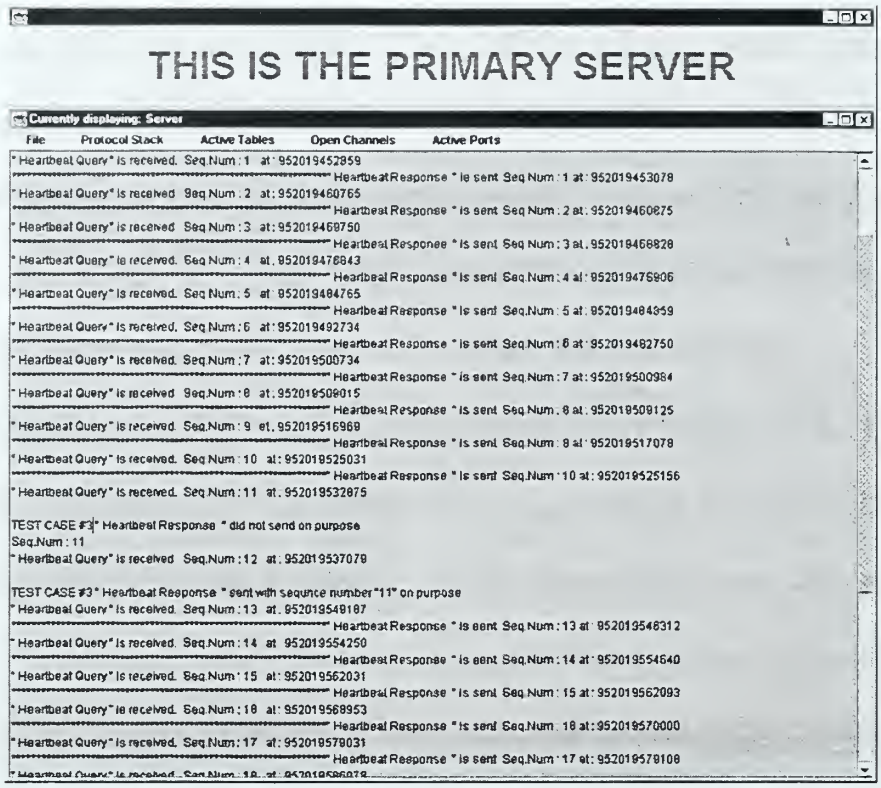

THIS IS THE PRIMARY SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query " is received. Seq Num : 1 at : 952019452859
" Heartbeat Response " is sent Seq Num : 1 at : 952019453078
" Heartbeat Query " is received. Seq Num : 2 at : 952019460765
" Heartbeat Response " is sent Seq Num : 2 at : 952019460875
" Heartbeat Query " is received. Seq Num : 3 at : 952019468750
" Heartbeat Response " is sent Seq Num : 3 at : 952019468828
" Heartbeat Query " is received. Seq Num : 4 at : 952019476843
" Heartbeat Response " is sent Seq Num : 4 at : 952019476906
" Heartbeat Query " is received. Seq Num : 5 at : 952019484765
" Heartbeat Response " is sent Seq Num : 5 at : 952019484859
" Heartbeat Query " is received. Seq Num : 6 at : 952019492734
" Heartbeat Response " is sent Seq Num : 6 at : 952019492750
" Heartbeat Query " is received. Seq Num : 7 at : 952019500734
" Heartbeat Response " is sent Seq Num : 7 at : 952019500984
" Heartbeat Query " is received. Seq Num : 8 at : 952019509015
" Heartbeat Response " is sent Seq Num : 8 at : 952019509125
" Heartbeat Query " is received. Seq Num : 9 at : 952019516969
" Heartbeat Response " is sent Seq Num : 9 at : 952019517078
" Heartbeat Query " is received. Seq Num : 10 at : 952019525031
" Heartbeat Response " is sent Seq Num : 10 at : 952019525156
" Heartbeat Query " is received. Seq Num : 11 at : 952019532875
TEST CASE #3 " Heartbeat Response " did not send on purpose
Seq.Num : 11
" Heartbeat Query " is received. Seq.Num : 12 at : 952019537078
TEST CASE #3 " Heartbeat Response " sent with sequnce number "11" on purpose
" Heartbeat Query " is received. Seq Num : 13 at : 952019548187
" Heartbeat Response " is sent Seq Num : 13 at : 952019548312
" Heartbeat Query " is received. Seq Num : 14 at : 952019554250
" Heartbeat Response " is sent Seq Num : 14 at : 952019554640
" Heartbeat Query " is received. Seq Num : 15 at : 952019562031
" Heartbeat Response " is sent Seq Num : 15 at : 952019562093
" Heartbeat Query " is received. Seq Num : 16 at : 952019568953
" Heartbeat Response " is sent Seq Num : 16 at : 952019570000
" Heartbeat Query " is received. Seq Num : 17 at : 952019579031
" Heartbeat Response " is sent Seq Num : 17 at : 952019579108

Figure 5.36. The Primary Server GUI in Message Numbering Scheme Test (Step 1).

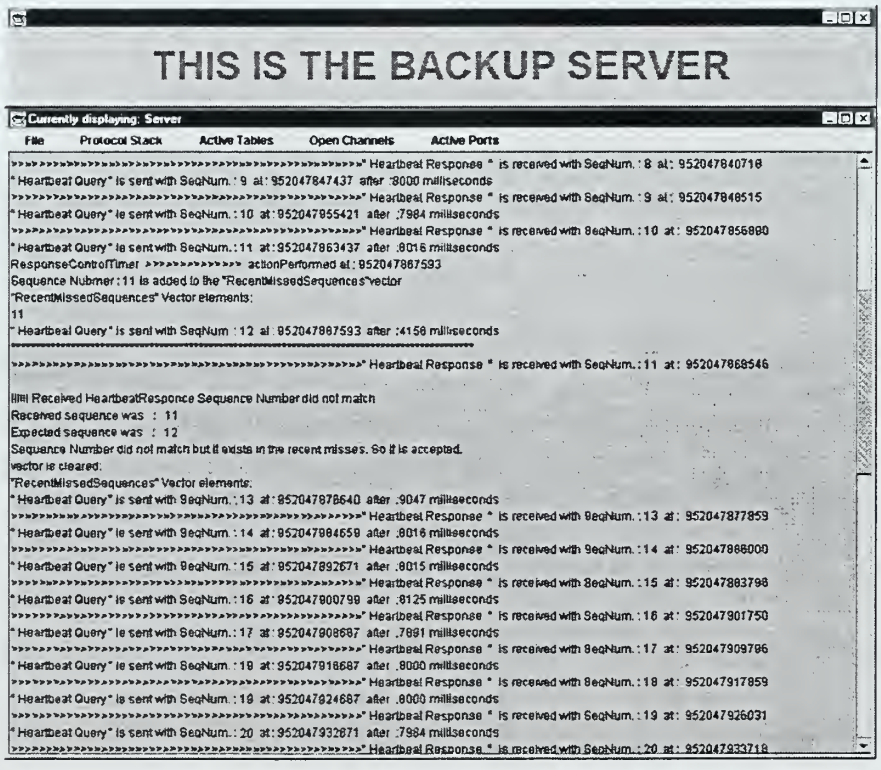

THIS IS THE BACKUP SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Response " is received with SeqNum. : 8 at : 952047840718
" Heartbeat Query " is sent with SeqNum. : 9 at : 952047847437 after :8000 milliseconds
" Heartbeat Response " is received with SeqNum. : 9 at : 952047848515
" Heartbeat Query " is sent with SeqNum. : 10 at : 952047855421 after :7984 milliseconds
" Heartbeat Response " is received with SeqNum. : 10 at : 952047856880
" Heartbeat Query " is sent with SeqNum. : 11 at : 952047863437 after :8016 milliseconds
ResponseControlTimer >>>>>>>>>>>>>> actionPerformed at : 952047867593
Sequence Nubmer : 11 is added to the "RecentMissedSequences" vector
"RecentMissedSequences" Vector elements:
11
" Heartbeat Query " is sent with SeqNum. : 12 at : 952047867593 after :4156 milliseconds
" Heartbeat Response " is received with SeqNum. : 11 at : 952047868546
!!!!! Received HeartbeatResponce Sequence Number did not match
Received sequence was : 11
Expected sequence was : 12
Sequence Number did not match but it exists in the recent misses. So it is accepted.
vector is cleared.
"RecentMissedSequences" Vector elements:
" Heartbeat Query " is sent with SeqNum. : 13 at : 952047878640 after :9047 milliseconds
" Heartbeat Response " is received with SeqNum. : 13 at : 952047877859
" Heartbeat Query " is sent with SeqNum. : 14 at : 952047884659 after :8016 milliseconds
" Heartbeat Response " is received with SeqNum. : 14 at : 952047886000
" Heartbeat Query " is sent with SeqNum. : 15 at : 952047892671 after :8015 milliseconds
" Heartbeat Response " is received with SeqNum. : 15 at : 952047893798
" Heartbeat Query " is sent with SeqNum. : 16 at : 952047900799 after :8125 milliseconds
" Heartbeat Response " is received with SeqNum. : 16 at : 952047901750
" Heartbeat Query " is sent with SeqNum. : 17 at : 952047908687 after :7891 milliseconds
" Heartbeat Response " is received with SeqNum. : 17 at : 952047909796
" Heartbeat Query " is sent with SeqNum. : 18 at : 952047916687 after :8000 milliseconds
" Heartbeat Response " is received with SeqNum. : 18 at : 952047917859
" Heartbeat Query " is sent with SeqNum. : 19 at : 952047924687 after :8000 milliseconds
" Heartbeat Response " is received with SeqNum. : 19 at : 952047926031
" Heartbeat Query " is sent with SeqNum. : 20 at : 952047932671 after :7984 milliseconds
" Heartbeat Response " is received with SeqNum. : 20 at : 952047933718

Figure 5.37. The Backup Server GUI in Message Numbering Scheme Test (Step 1).

In the second step of the test, the primary server did not respond to the heartbeat query messages that had the 11th and 12th sequence numbers (see Figure 5.38). However, the primary server responded to the 13th heartbeat query message with the heartbeat response message that had sequence number three. In this case, the backup server could not find the "three" in the `recentMissedSequences` vector and ignored the heartbeat response message (see Figure 5.39).

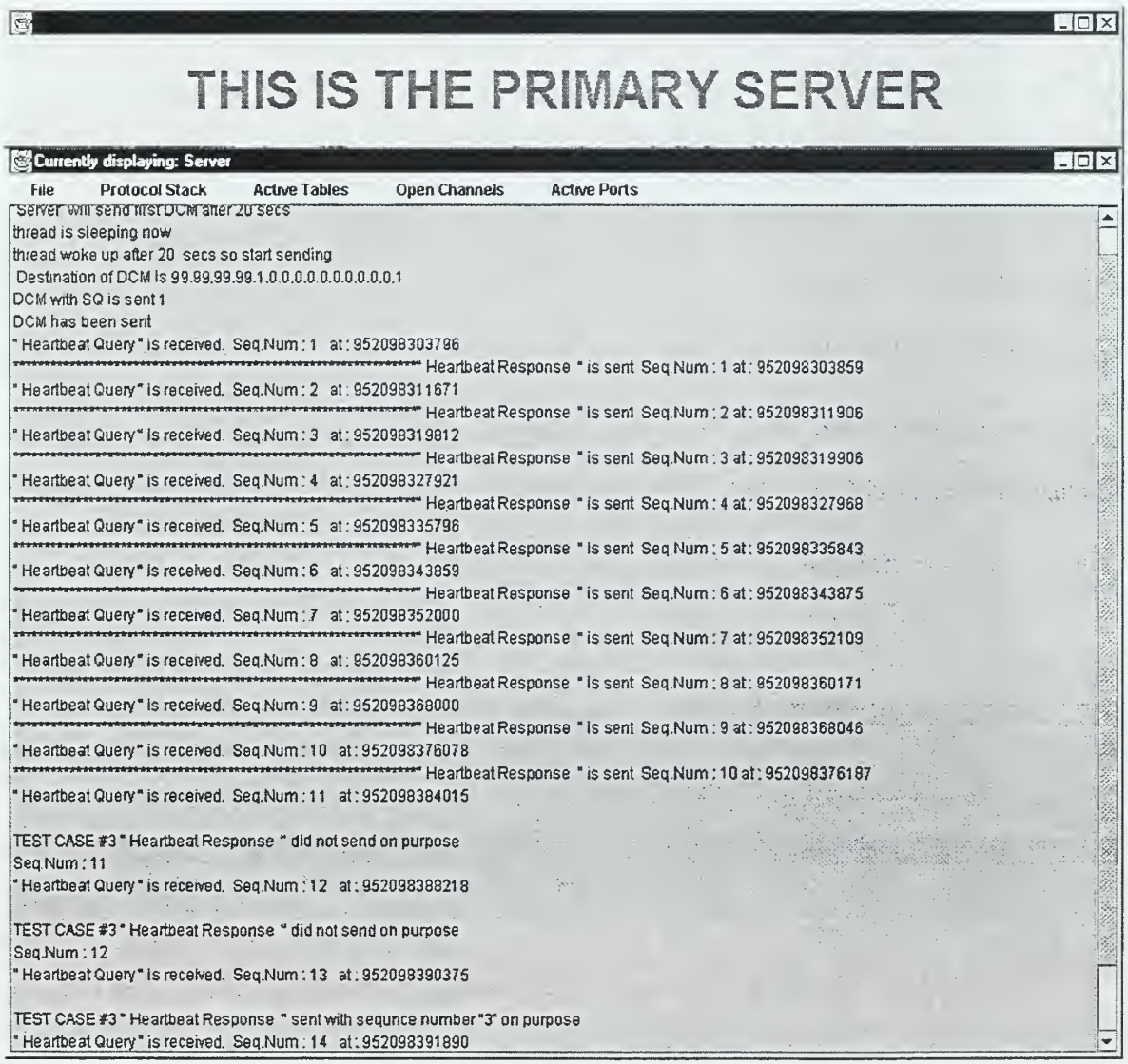
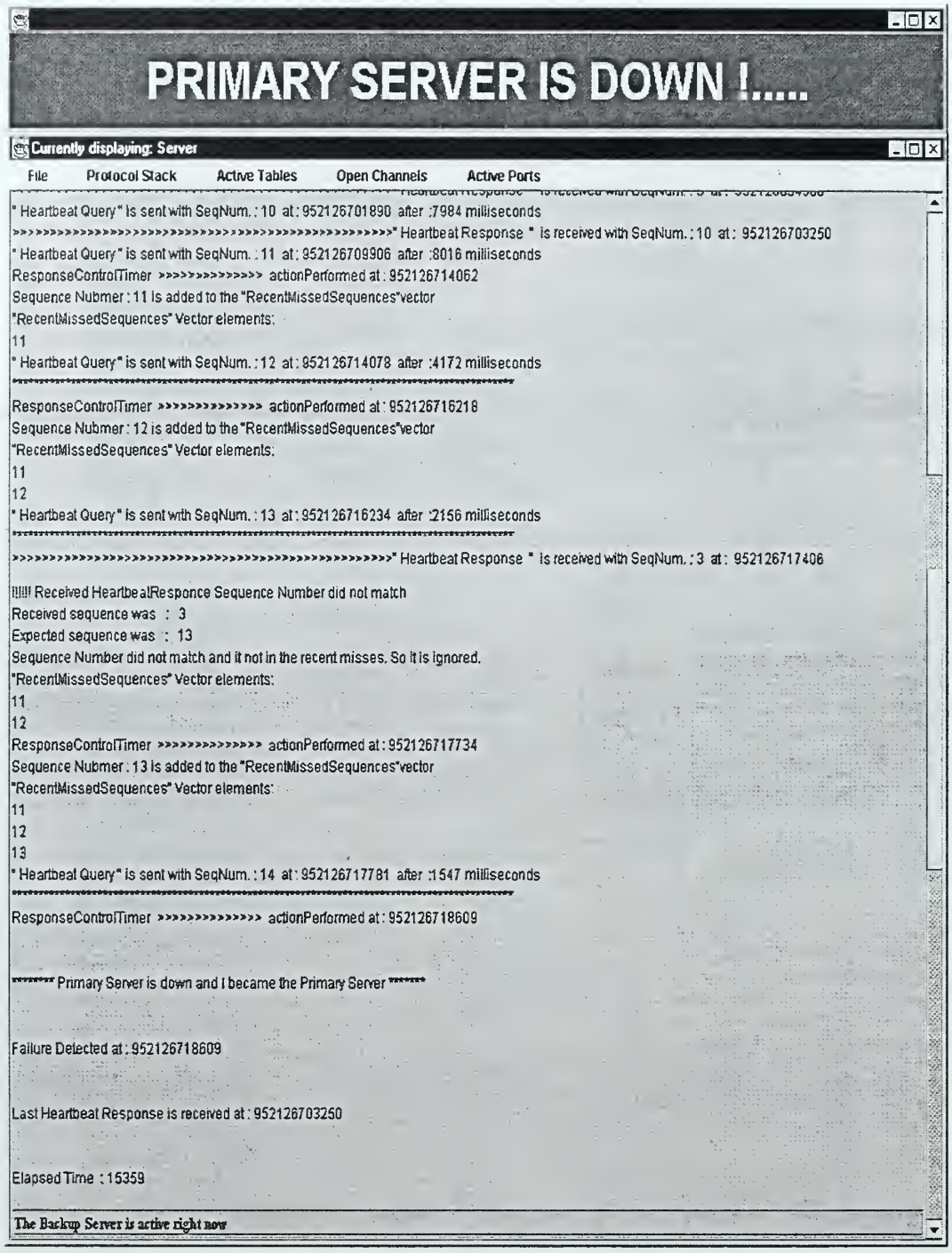

THIS IS THE PRIMARY SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
Server will send first DCM after 20 secs
thread is sleeping now
thread woke up after 20 secs so start sending
Destination of DCM is 99.99.99.98.1.0.0.0.0.0.0.0.0.0.0.1
DCM with SQ is sent 1
DCM has been sent
" Heartbeat Query " is received. Seq.Num : 1 at : 952098303796
********** Heartbeat Response " is sent Seq.Num : 1 at : 952098303859
" Heartbeat Query " is received. Seq.Num : 2 at : 952098311671
********** Heartbeat Response " is sent Seq.Num : 2 at : 952098311906
" Heartbeat Query " is received. Seq.Num : 3 at : 952098319812
********** Heartbeat Response " is sent Seq.Num : 3 at : 952098319906
" Heartbeat Query " is received. Seq.Num : 4 at : 952098327921
********** Heartbeat Response " is sent Seq.Num : 4 at : 952098327968
" Heartbeat Query " is received. Seq.Num : 5 at : 952098335796
********** Heartbeat Response " is sent Seq.Num : 5 at : 952098335843
" Heartbeat Query " is received. Seq.Num : 6 at : 952098343859
********** Heartbeat Response " is sent Seq.Num : 6 at : 952098343875
" Heartbeat Query " is received. Seq.Num : 7 at : 952098352000
********** Heartbeat Response " is sent Seq.Num : 7 at : 952098352109
" Heartbeat Query " is received. Seq.Num : 8 at : 952098360125
********** Heartbeat Response " is sent Seq.Num : 8 at : 952098360171
" Heartbeat Query " is received. Seq.Num : 9 at : 952098368000
********** Heartbeat Response " is sent Seq.Num : 9 at : 952098368046
" Heartbeat Query " is received. Seq.Num : 10 at : 952098376078
********** Heartbeat Response " is sent Seq.Num : 10 at : 952098376187
" Heartbeat Query " is received. Seq.Num : 11 at : 952098384015
TEST CASE #3 " Heartbeat Response " did not send on purpose
Seq.Num : 11
" Heartbeat Query " is received. Seq.Num : 12 at : 952098388218
TEST CASE #3 " Heartbeat Response " did not send on purpose
Seq.Num : 12
" Heartbeat Query " is received. Seq.Num : 13 at : 952098390375
TEST CASE #3 " Heartbeat Response " sent with sequnce number "3" on purpose
" Heartbeat Query " is received. Seq.Num : 14 at : 952098391890

Figure 5.38. The Primary Server GUI in Message Numbering Scheme Test (Step 2).


PRIMARY SERVER IS DOWN !.....
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query " is sent with SeqNum. : 10 at : 952126701890 after :7984 milliseconds
>>>

During the test we observed that the backup server was handling the message sequence numbers as specified in the implementation design. Additionally, the backup server was utilizing the `recentMissedSequences` vector correctly. Consequently, the implementation successfully passed the message numbering scheme test.

d. Unsolicited Heartbeat Test

In order to prevent a false failure detection, the backup server always performs an additional check for new DCM message before declaring the failure of the primary server. More specifically, even though the backup server does not receive heartbeat responses, if the backup server receives a new DCM message from the primary server during the current period, it sets the next period to t_{max}. The unsolicited heartbeat test (`testCase4`) was performed in order to test such usage of the DCM messages in the failure detection process.

In this test, the values of the C (DCM message interval), the t_{max} and t_{min} were 14.0, 16.0, and 1.0 seconds, respectively. To perform the test, only the `testCase4` was set to "true" among the test case booleans of the `Server` class. During the test run, the primary server did not respond to the heartbeat query messages with the sequence numbers of 5, 6, 7, and 8 (see Figure 5.40). After that, the primary server correctly responded to the heartbeat query messages. However, although the backup server did not receive four consecutive heartbeat responses from the primary server, it detected that the DCM messages were still coming. Therefore, instead of

declaring the failure of the main server, the backup server made the length of the next period t_{max} (see Figure 5.41).

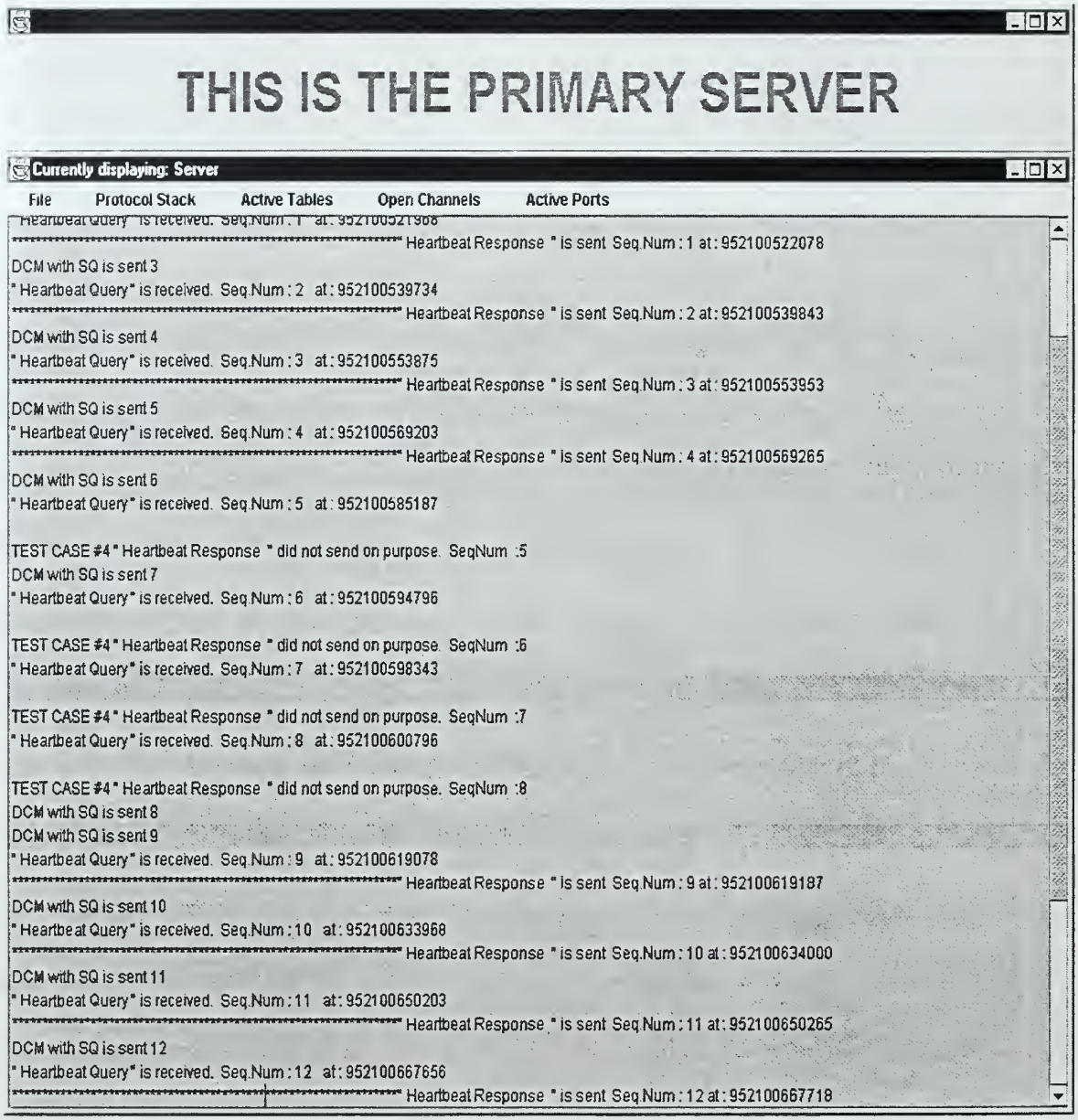
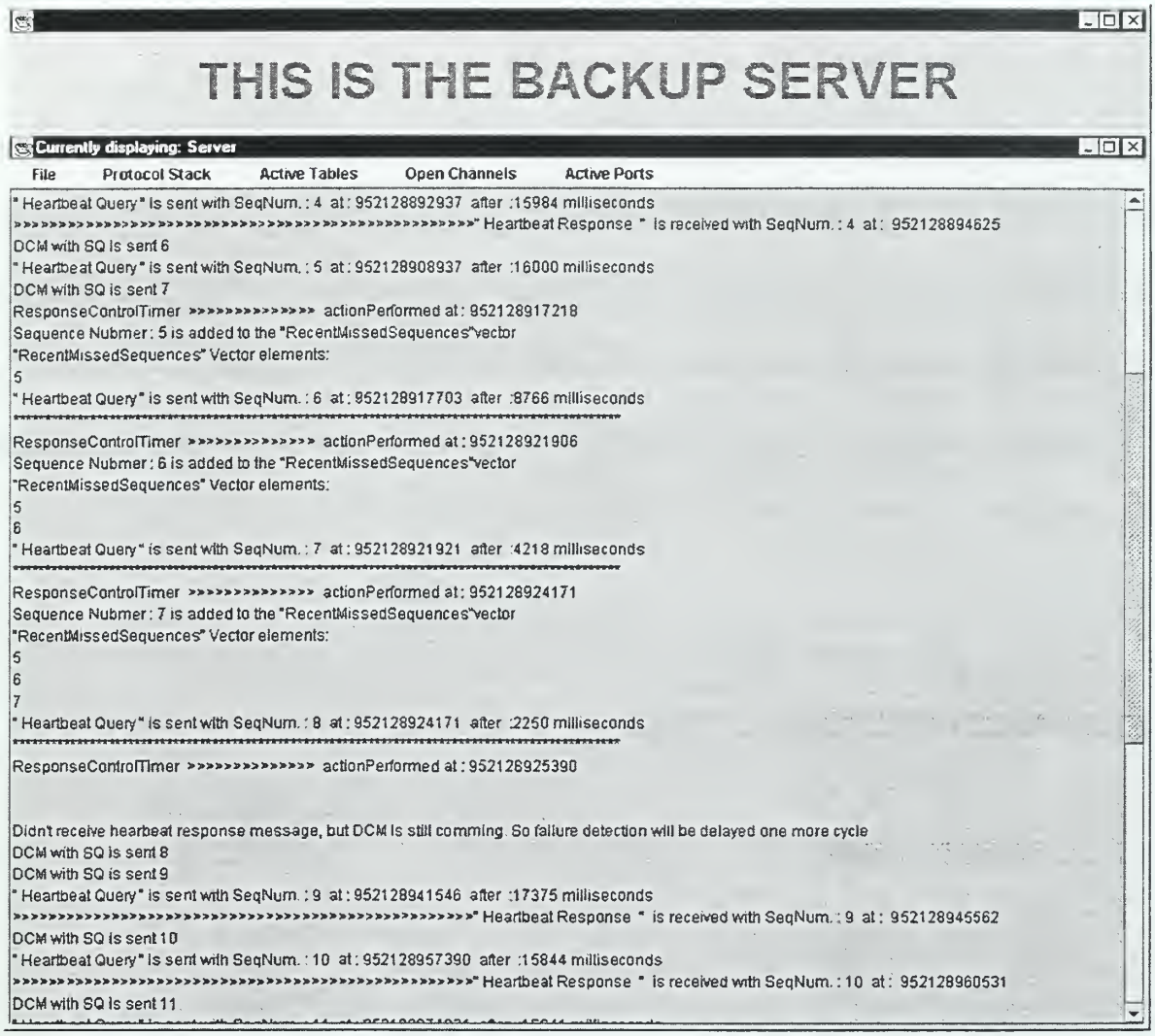

THIS IS THE PRIMARY SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Response " is sent Seq.Num : 1 at : 952100522078
DCM with SQ is sent 3
" Heartbeat Query" is received. Seq.Num : 2 at : 952100539734
" Heartbeat Response " is sent Seq.Num : 2 at : 952100539843
DCM with SQ is sent 4
" Heartbeat Query" is received. Seq.Num : 3 at : 952100553875
" Heartbeat Response " is sent Seq.Num : 3 at : 952100553953
DCM with SQ is sent 5
" Heartbeat Query" is received. Seq.Num : 4 at : 952100569203
" Heartbeat Response " is sent Seq.Num : 4 at : 952100569265
DCM with SQ is sent 6
" Heartbeat Query" is received. Seq.Num : 5 at : 952100585187
TEST CASE #4 " Heartbeat Response " did not send on purpose. SeqNum :5
DCM with SQ is sent 7
" Heartbeat Query" is received. Seq.Num : 6 at : 952100594796
TEST CASE #4 " Heartbeat Response " did not send on purpose. SeqNum :6
" Heartbeat Query" is received. Seq.Num : 7 at : 952100598343
TEST CASE #4 " Heartbeat Response " did not send on purpose. SeqNum :7
" Heartbeat Query" is received. Seq.Num : 8 at : 952100600796
TEST CASE #4 " Heartbeat Response " did not send on purpose. SeqNum :8
DCM with SQ is sent 8
DCM with SQ is sent 9
" Heartbeat Query" is received. Seq.Num : 9 at : 952100619078
" Heartbeat Response " is sent Seq.Num : 9 at : 952100619187
DCM with SQ is sent 10
" Heartbeat Query" is received. Seq.Num : 10 at : 952100633968
" Heartbeat Response " is sent Seq.Num : 10 at : 952100634000
DCM with SQ is sent 11
" Heartbeat Query" is received. Seq.Num : 11 at : 952100650203
" Heartbeat Response " is sent Seq.Num : 11 at : 952100650265
DCM with SQ is sent 12
" Heartbeat Query" is received. Seq.Num : 12 at : 952100667656
" Heartbeat Response " is sent Seq.Num : 12 at : 952100667718

Figure 5.40. The Primary Server GUI in Unsolicited Heartbeat Test.


THIS IS THE BACKUP SERVER
Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
" Heartbeat Query " is sent with SeqNum. : 4 at : 952128892937 after :15984 milliseconds
>>>

5.5). The control channel auto configuration test was performed to test the functionality of the aforementioned relationship between the t_{max} and C.

In this test, the values of the C, the t_{max} and the t_{min} were 70.0, 80.0, and 2.5 seconds, respectively. After the test topology shown in Figure 5.30 was initialized by the demo station, the control channel for the primary SAAM server was built by the auto configuration mechanism. The dashed lines in Figure 5.42 show part of the control channel used for the heartbeat message exchange by the servers. The heartbeat response messages were reaching the backup server via the routers A, D, and E.

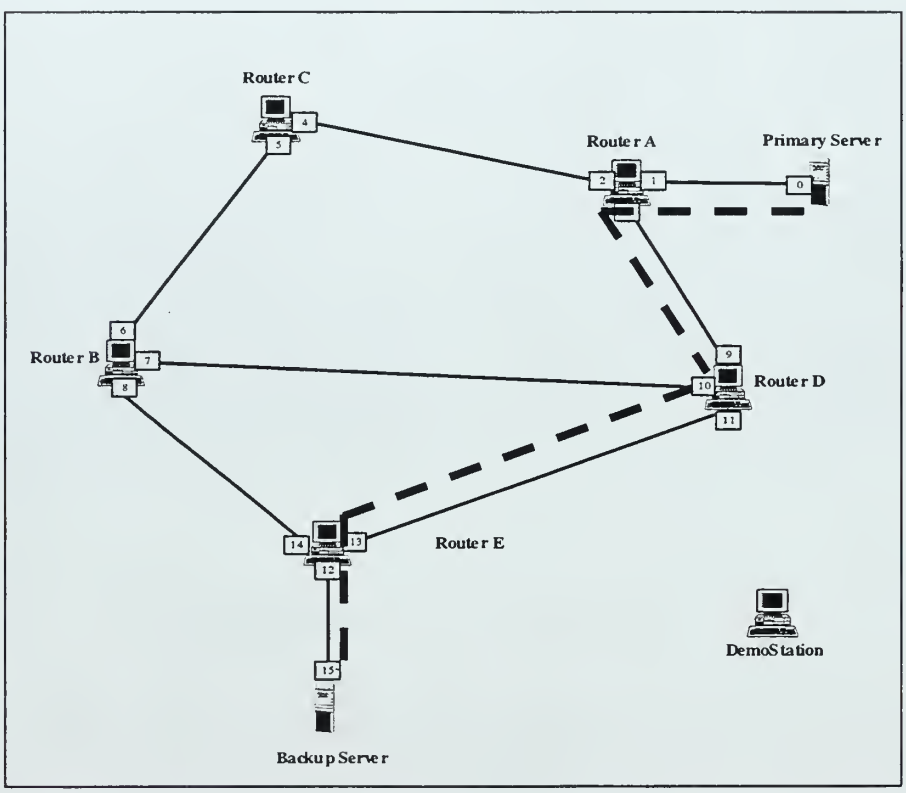


Figure 5.42. The Control Channel Path Used by the Heartbeat Response Messages.

To simulate the failure of the router D, the router application on that machine was manually killed. After the router D had failed, the heartbeat response messages with the sequence numbers 11, 12, 13, and 14 could not reach the backup server. However, the heartbeat response message with the sequence number 15 was able to reach the backup server via the newly constructed control channel (see Figure 5.43) within the 15th DCM cycle (see Figure 5.44).

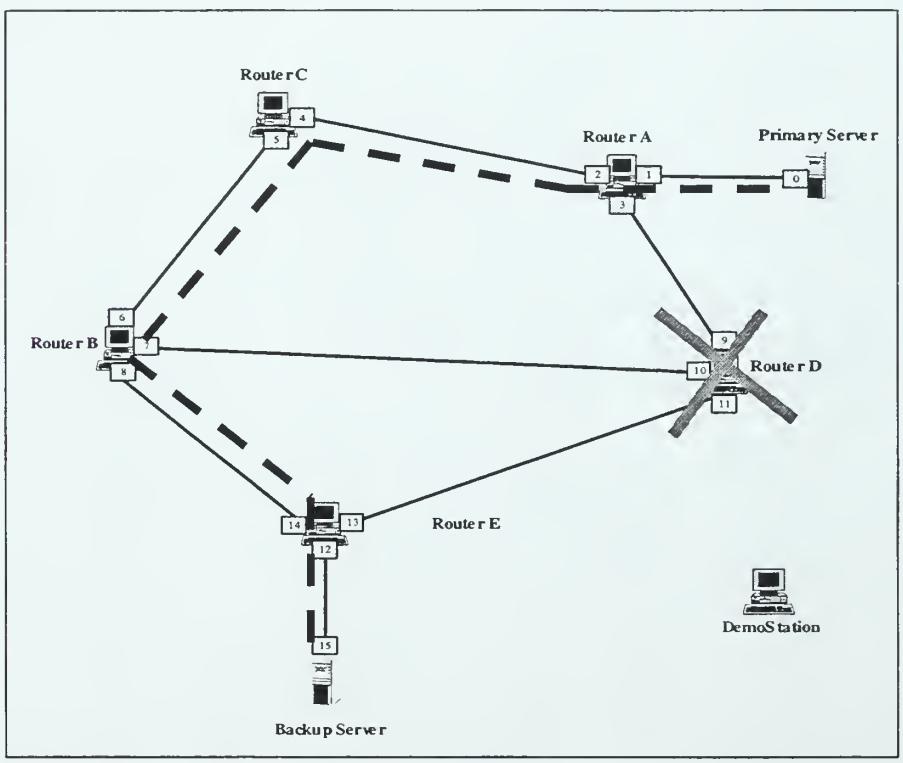
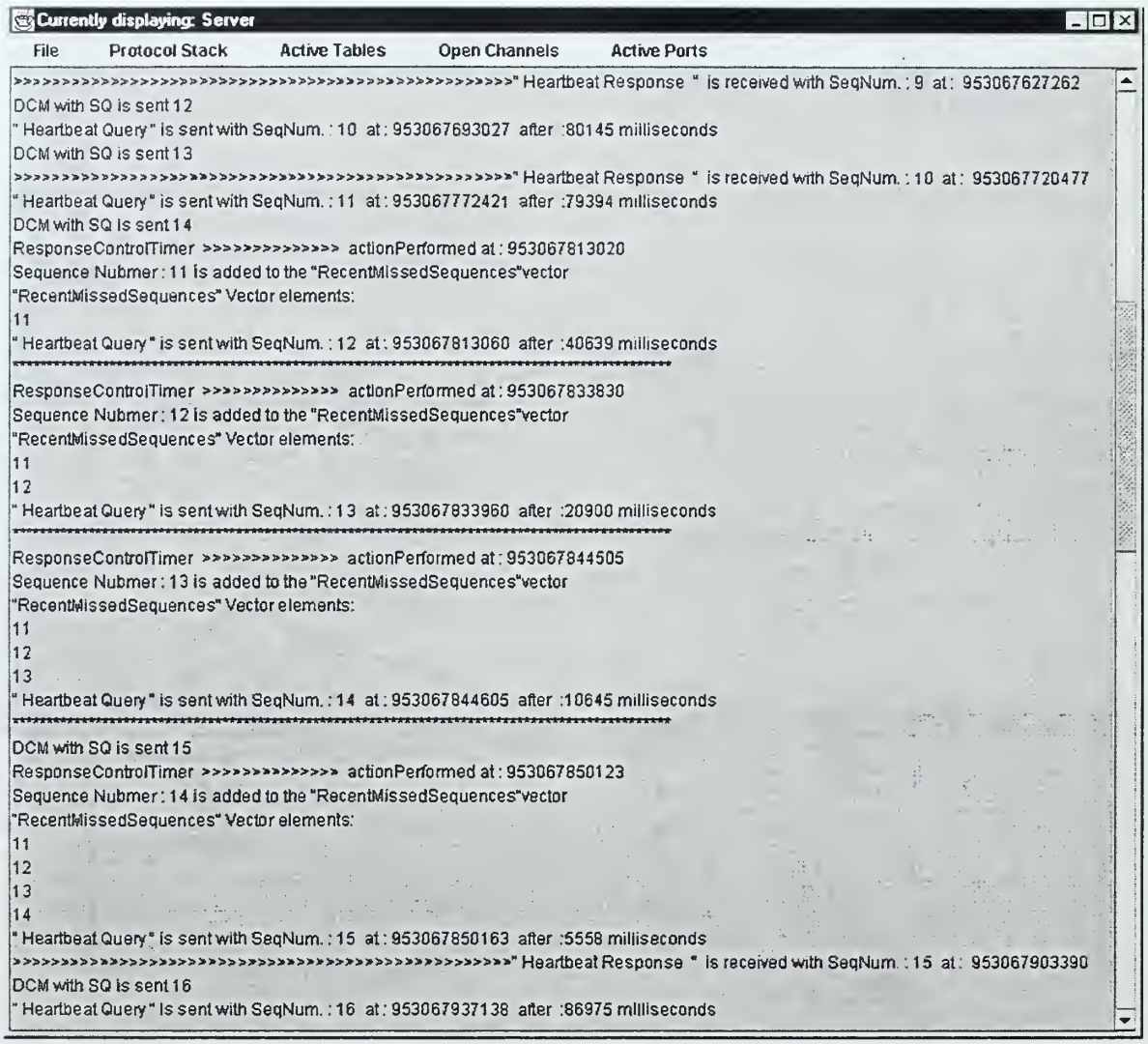


Figure 5.43. Reconstructed Control Channel Path After Failure of the Router D.

During the test, the control channel auto configuration mechanism of the SAAM server successfully reconstructed the control channel preventing false failure detection from happening. The test performed showed that as long as the relationship

between the t_{max} and the C given by Equation 5.5 is provided, failures of routers that are part of the control channel will not result in false failure detection. Consequently, the implementation successfully passed the control channel auto-configuration test.


Currently displaying: Server
File
Protocol Stack
Active Tables
Open Channels
Active Ports
>>>

VI. CONCLUSIONS

A. SYNOPSIS AND CONCLUSION

The main purpose of this thesis is to add fault tolerance features to the SAAM architecture so that server failures can be tolerated. As a result, a failure of the SAAM server will not interrupt SAAM services to the routers.

In order to provide a fault tolerance solution for the SAAM server that best meets the requirements mentioned in Chapter I, fault tolerance for the SAAM servers is examined in two phases: local and remote. The first phase, local area fault tolerance for the SAAM server, focuses mainly on tolerating the component failures of one server. The second phase, remote area fault tolerance (disaster recovery) for the SAAM server, focuses on tolerating environmental faults that cause unrecoverable server failures.

For local area fault tolerance, a COTS-based solution is proposed after a survey of the literature and commercial offerings. The proposed solution is based on Endurance 4000, which can recover from server failures in less than a second. Additionally, the systems connected by Endurance 4000 can be placed at different locations up to 1.5 kilometers apart, providing fault tolerance for the environmental faults to some degree. For these reasons, especially its ability to recover from server failures in milliseconds, we believe that Endurance 4000 best meets the criteria for local area fault tolerance for the SAAM server.

A backup server model is designed and implemented for remote area fault tolerance. Although the designed model covers all aspects of remote area fault tolerance for the SAAM server, the implementation only covers the failure detection, damage

confinement and assessment, and failure recovery phases. The prototyped model provides robust error detection and a fast recovery from a failure of the primary SAAM server.

During the design process, two different heartbeat protocols were prototyped and their performance results were compared to determine the best failure detection mechanism for the SAAM server.

In the end, the accelerated heartbeat protocol was selected and implemented as the failure detection mechanism for the remote area fault tolerance. Then, the prototype was integrated with the existing server source code and the whole system was tested in a live SAAM testbed.

In order to evaluate the functionality of the remote area fault tolerance implementation for the SAAM server five different test conditions are tested. At the end of each test, we observed that the remote area fault tolerance implementation for the SAAM server behaved in accordance with its specifications. Therefore, the integration of the prototyped model with the existing server source code was successful.

The remote area fault tolerance approach introduced in this thesis, demonstrated a robust failure detection capability and a fast recovery from the failure of the primary SAAM server. With these specifications, the remote area fault tolerance approach proposed in this thesis applies not only to SAAM server but also to any military application that requires timely recovery from the environmental faults that might occur.

B. FUTURE WORK

This thesis is only an initial effort to add fault tolerant features to the SAAM server given the time constraints. As such, only the failure detection, damage

confinement and assessment, and failure recovery phases of the remote area fault tolerance design are implemented. There are numerous open issues. In the following sections, these issues are outlined.

1. Testing of Recommended COTS-Based Product

In this thesis, for local area fault tolerance, a COTS-based solution (Endurance 4000) is proposed after a survey of the literature and commercial offerings. According to its specifications, we believe that Endurance 4000 best meets the local area fault tolerance requirements of the SAAM server. However, the actual performance of this product should be tested in a live SAAM test bed and the test results should be evaluated.

2. Detection of Backup SAAM Server Failures

In the remote area fault tolerance implementation, this thesis focused only on the detection of failures the primary SAAM server. The backup SAAM server is assumed to always be ready to take over in the event of such failures. However, the backup SAAM server can also fail while in its silent running state. In this case, a failure of the primary SAAM server can no longer be tolerated. Therefore, the failure of the backup SAAM server should also be detected and corrected as early as possible. For this purpose, a failure detection mechanism for the backup SAAM server should be added to the current remote area fault tolerance implementation.

3. Reinstating of a Repaired Server

When the primary SAAM server fails and the backup SAAM server takes over the job of the primary server, the backup SAAM server is no longer fault tolerant. Therefore, the failed server should be repaired and reinstated as soon as possible.

Although some details are given in the modeling section, the process of reinstating the repaired server is not covered in the current remote area fault tolerance implementation. However, reinstating a repaired server is an essential phase of remote area fault tolerance and this functionality must be added in the feature.

4. Handling of Two Simultaneously Active Servers

The proposed measures for preventing false failure detection should dramatically reduce the probability of two servers being in the active running state at the same time. However, the probability of two servers simultaneously in the active running state is not zero. Therefore, such a case must be considered and handled by the system. In the modeling section of the remote area fault tolerance for the SAAM server, some details on how to handle the existence of two simultaneously active servers are provided. However, this case is not covered in the current remote area fault tolerance implementation.

5. Field Test

In this thesis, the implementation of remote area fault tolerance is only tested for its basic functionality. Due to limitations of the emulated environment, real performance data could not be collected. In our tests, largely scaled t_{max} and t_{min} values are used for the time interval between the heartbeat query messages and for the round trip delay upper

bound between two servers. However, the best values for the t_{max} and the t_{min} should be determined by performing tests on a real (non-emulated) SAAM testbed.

6. Alert Mechanism

In order to speed up the repairing process, the administrator should be immediately notified when a SAAM server fails. For this purpose, an alert mechanism should be added to the current implementation. An alert mechanism can be in the form of a message displayed on the screen, an e-mail/voice message to the administrator, a sound alarm, or some combination of these alert types.

In the current implementation of remote area fault tolerance for the SAAM server, the failure of the primary SAAM server is displayed on the backup SAAM server screen. However, this basic alert mechanism requires an administrative person in front of the backup SAAM server terminal, which is not practical. Therefore, a robust alert mechanism should be added to the remote area fault tolerance implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. THE CONSTANT HEARTBEAT PROTOCOL SOURCE FILES

```
//---------------------------------------------------------------------------
// Filename:       PrimaryServer.java
// Date     :      October 23, 1999
// Name     :      Efraim KATI
// Project  :      Constant Heartbeat Protocol (V.0)
// Compiler:       JDK 1.2.2
//---------------------------------------------------------------------------

package constantheartbeat;

import java.awt.*;
import java.awt.event.*;

/**
 * This class represents the Primary Server side of the "Constant Heartbeat"
 * protocol. In this prototype implementation, Primary Server periodically
 * sends "I am alive" messages (Heartbeats) to the Backup Server. This class
 * first displays a GUI. User can select the heartbeat message interval
 * (in seconds) from the drop-down choice menu.  It has a text area to display
 * the messages communicated between servers. When the user presses the start
 * button, provided in the GUI, this class creates a new thread which sends
 * the heartbeat messages periodically
 *
 * @author Efraim KATI
 */

public class PrimaryServer extends Frame {

  /**
   * primaryLabel is a Label object. It is used for displaying "PRIMARY
   * SERVER" text on the window.
   */

   private Label primaryLabel;

  /**
   * display is a TextArea object. It is used for displaying messages.
   */

   private TextArea display;

  /**
   * startButton is a Button object. It starts sending heartbeat messages.
   */

   private Button startButton;

  /**
   * stopButton is a Button object. It stops sending heartbeat messages.
   */
```

```
 private Button stopButton;

/**
 * intervalChoice is a Choice object. It is used for selecting the
 * interval value from drop-down list.
 */

 private Choice intervalChoice;

/**
 * intervalLabel is a Label object. It is used for displaying "SELECT TIME
 * INTERVAL FOR HEARTBEAT MESSAGES (sec.)" text on the window.
 */

 private Label intervalLabel;

/**
 * primaryServerThread is a PrimaryServerThread object
 */

 private PrimaryServerThread primaryServerThread;

/**
 * exitButton is a Button object. When clicked, exits the program.
 */

 private Button exitButton;

/**
 * interval is an integer that is used for heartbeat message sending
 * interval
 */

 private double interval;

/**
 * isAlive is a boolean. When it is true, shows that primary server is
 * running
 */

 private boolean isAlive;


/**
 * Constructor for the PrimaryServer class. It initializes and sets the
 * specifications of the data members.
 *
 * @param none
 */

 public PrimaryServer() {

    try  {
```

```
        //initialize the GUI components
        jbInit();
        //initialize other data members
        isAlive = false;
        interval = 0.5;

    }
    catch(Exception e) {

        display.append(e.getMessage());

    }//end try/catch

 }//end PrimaryServer()


/**
 * This method is the main method of the PrimaryServer class. It creates a
 * PrimaryServer object.
 *
 * @param args String[]
 * @return void
 */

 public static void main(String[] args) {

    PrimaryServer primaryServer = new PrimaryServer();

 }//end main()

/**
 * Returns the isAlive boolean value.
 *
 * @param none
 * @return isAlive
 */

 public boolean isAlive(){

    return isAlive;

 }//end isAlive()

/**
 * This method initializes GUI components of the PrimaryServer class.
 *
 * @param none
 * @return void
 * @exception Exception
 */

 private void jbInit() throws Exception {

    //create GUI components
    primaryLabel = new Label();
```

```
display = new TextArea();
startButton = new Button();
stopButton = new Button();
intervalChoice = new Choice();
intervalLabel = new Label();
exitButton = new Button();

//initialize primaryLabel
primaryLabel.setBackground(Color.lightGray);
primaryLabel.setBounds(new Rectangle(34, 28, 436, 48));
primaryLabel.setFont(new java.awt.Font("SansSerif", 1, 34));
primaryLabel.setForeground(Color.black);
primaryLabel.setAlignment(1);
primaryLabel.setText("PRIMARY SERVER");

//initialize display
display.setBackground(Color.white);
display.setBounds(new Rectangle(14, 78, 486, 364));
display.setFont(new java.awt.Font("Dialog", 1, 14));
display.append("\n\n                                    INSTRUCTIONS\n");
display.append("\n**************************************************"+
               "***************");
display.append("\n\n                  1. SELECT THE TIME INTERVAL ");
display.append("\n\n                  2. PRESS START BUTTON");
display.append("\n\n                  3. WHEN YOU WANT TO STOP SENDING"+
               "HEARTBEAT\n"+
              "                      MESSAGES, PRESS STOP BUTTON");
display.setEditable(false);

//initialize startButton
startButton.setBackground(Color.lightGray);
startButton.setBounds(new Rectangle(146, 457, 105, 36));
startButton.setFont(new java.awt.Font("Dialog", 1, 20));
startButton.setLabel("START");
startButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      startButton_actionPerformed(e);
   }
});

//initialize stopButton
stopButton.setLabel("STOP");
stopButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      stopButton_actionPerformed(e);
   }
});
stopButton.setFont(new java.awt.Font("Dialog", 1, 20));
stopButton.setBackground(Color.lightGray);
stopButton.setBounds(new Rectangle(258, 457, 105, 36));

//initialize intervalLabel
intervalLabel.setBackground(Color.lightGray);
intervalLabel.setBounds(new Rectangle(12, 497, 447, 36));
intervalLabel.setFont(new java.awt.Font("Dialog", 1, 15));
```

```
    intervalLabel.setForeground(Color.black);
    intervalLabel.setAlignment(1);
    intervalLabel.setText("SELECT TIME INTERVAL FOR HEARTBEAT " +
                          "MESSAGES (sec.)");
    //initialize exitButton
    exitButton.setBackground(Color.lightGray);
    exitButton.setBounds(new Rectangle(205, 553, 104, 35));
    exitButton.setFont(new java.awt.Font("Dialog", 1, 20));
    exitButton.setForeground(Color.black);
    exitButton.setLabel("EXIT");
    exitButton.addActionListener(new java.awt.event.ActionListener() {
       public void actionPerformed(ActionEvent e) {
         exitButton_actionPerformed(e);
       }
    });

    //initialize intervalChoice
    intervalChoice.addItemListener(new java.awt.event.ItemListener() {
       public void itemStateChanged(ItemEvent e) {
         intervalChoice_itemStateChanged(e);
       }
    });
    intervalChoice.setBounds(new Rectangle(463, 500, 44, 28));
    intervalChoice.setFont(new java.awt.Font("Dialog", 1, 15));
    intervalChoice.add("0.5");
    intervalChoice.add("1");
    intervalChoice.add("1.5");
    intervalChoice.add("2");
    intervalChoice.add("2.5");
    intervalChoice.add("3");

    //initialize frame properties
    this.setBackground(Color.lightGray);
    this.setEnabled(true);
    this.setTitle("Constant Heartbeat Protocol (V.0)");
    this.addWindowListener(new java.awt.event.WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          this_windowClosing(e);
       }
    });
    this.setLayout(null);
    this.add(display, null);
    this.add(intervalLabel, null);
    this.add(intervalChoice, null);
    this.add(primaryLabel, null);
    this.add(stopButton, null);
    this.add(startButton, null);
    this.add(exitButton, null);
    this.setSize(515,600);
    this.setVisible(true);

 }//end jbInit()

/**
 * If "X" is clicked on window, terminates the program.
```

```
 *
 * @param e WindowEvent
 * @return void
 */

 private void this_windowClosing(WindowEvent e) {

    System.exit(1);

 }//end this_windowClosing()


/**
 * When startButton is pressed, clears the text area and starts the
 * PrimaryServerThread for sending heartbeat messages
 *
 * @param e ActionEvent
 * @return void
 */

 private void startButton_actionPerformed(ActionEvent e) {

    isAlive = true;
    display.setText("");
    primaryServerThread = new PrimaryServerThread(display,interval,this);
    primaryServerThread.start();
    intervalChoice.setEnabled(false);

 }//end startButton_actionPerformed()

/**
 * When stopButton is pressed, stops sending heartbeat messages.
 *
 * @param e ActionEvent
 * @return void
 */

 private void stopButton_actionPerformed(ActionEvent e) {

    display.append("\n\nPRIMARY SERVER STOPPED SENDING MESSAGES AT :  "+
                   System.currentTimeMillis() + "");
    isAlive = false;
    startButton.setEnabled(false);
    stopButton.setEnabled(false);

 }//end stopButton_actionPerformed()


/**
 * If exitButton is clicked, terminates the program.
 *
 * @param e ActionEvent
 * @return void
 */
```

```
    private void exitButton_actionPerformed(ActionEvent e) {

       System.exit(1);

    }//end exitButton_actionPerformed()


   /**
    * When a new value is selected for the intervalChoice, sets the interval
    * value
    *
    * @param e ItemEvent
    * @return void
    */

    private void intervalChoice_itemStateChanged(ItemEvent e) {

       interval = Double.parseDouble((e.getItem().toString()));

    }//end intervalChoice_itemStateChanged()

}//end PrimaryServer class

//end file PrimaryServer.java
```

```
//-----------------------------------------------------------------------------
// Filename:      PrimaryServerThread.java
// Date     :     October 23, 1999
// Name     :     Efraim KATI
// Project :      Constant Heartbeat Protocol (V.0)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package constantheartbeat;

import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * This class runs as a separate thread and is created by the PrimaryServer
 * class, when the start button is pressed. PrimaryServerThread periodically
 * sends "I AM ALIVE" (Heartbeat) messages to the Backup Server.
 *
 * @author Efraim KATI
 */

public class PrimaryServerThread extends Thread{

  /**
   * display is a reference to a TextArea object. It is used for displaying
   * messages on the GUI.
   */

   private TextArea display;

  /**
   * primaryServer is a reference to the PrimaryServer object
   */

   private PrimaryServer primaryServer;

  /**
   * count is an integer and used for giving numbers to the heartbeat messages
   */

   private int count = 0;

  /**
   * interval is an integer that is used for heartbeat message sending
   * interval
   */

   private double interval;

  /**
   * outString is a String used for outgoing message
   */
```

```
 private String outString;

/**
 * outString is a String represents the heartbeat message
 */

 private String heartbeatMessage;


/**
 * Constructor for the PrimaryServerThread class. It initializes and sets
 * the specifications of the data members.
 *
 * @param none
 */

 public PrimaryServerThread(TextArea inDisplay,
                         double interval,
                         PrimaryServer primaryServer) {

    this.display = inDisplay;
    this.interval = interval;
    this.primaryServer = primaryServer;
    heartbeatMessage = "I AM ALIVE";

 }//end PrimaryServerThread()


/**
 * Run method of this thread. Message sending process is performed in this
 * method
 *
 * @param none
 * @return void
 */

 public void run(){

    //continue to send message, unless the stop button is pressed
    while(primaryServer.isAlive()){

       try{
          //create a socket
          Socket socket = new Socket(InetAddress.getLocalHost(), 8888);
          socket.setTcpNoDelay(true);
          OutputStream out =socket.getOutputStream();
          StringBuffer ack = new StringBuffer();
          ack.append(Double.toString(interval));
          ack.append("~");
          ack.append(heartbeatMessage);
          ack.append(" #" + count++ + "");
          outString=null;
          outString = ack.toString();
          display.append("\nSending.....:  \" " + outString +
                         " \"      at:  ");
```

```
            display.append(""+System.currentTimeMillis()+"");
            //send the message
            out.write(outString.getBytes());
            socket.close();
            //wait for one interval duration
            this.sleep((int)(interval*1000));

         }
         catch( Exception e ){

            display.append("\n" + e.getMessage());

         }//end try/catch

      }//end while

   }// end run()

}//end PrimaryServerThread class

//end file PrimaryServerThread.java
```

```
//-----------------------------------------------------------------------------
// Filename:      BackupServer.java
// Date     :     October 23, 1999
// Name     :     Efraim KATI
// Project :      Constant Heartbeat Protocol (V.0)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package constantheartbeat;

import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

/**
 * This class represents the Backup Server side of the "Constant Heartbeat"
 * protocol. In this prototype implementation, Primary Server periodically
 * emits "I am alive" messages (Heartbeats) to the Backup Server. Backup
 * Server listens these messages from the Primary Server. If predetermined
 * number of messages are missed, then Backup Server declares the failure of
 * the Primary Server
 *
 * Backup Server Class is implemented with GUI. It has a text area to display
 * messages. It has also dropdown menu, which gives user option to select the
 * the tolerated number of packet misses.
 *
 * @author Efraim KATI
 */

public class BackupServer extends Frame {


  /**
   * timer is a swing Timer Class object. Timer is restarted whenever a new
   * message is received. When the timer reaches to zero, it means that
   * Primary Server failed to send pre-determined number of messages to backup
   * server
   */

   protected Timer timer;

  /**
   * display is a TextArea object. It is used for displaying messages.
   */

   protected TextArea display;

  /**
   * CONV_MILLIS is a constant value equals to 1000. It is used for converting
   * seconds into milliseconds.
   */
   private final int CONV_MILLIS = 1000;
```

```
/**
 * timerHandler is a TimerHandler Class object. It listens the timer. When
 * the timer sends an actionPerformed()call, it changes the status boolean
 * value to false, which means Primary Server is down.
 */

 private TimerHandler timerHandler;

/**
 * serverSoc is a ServerSocket Class object. It waits for requests to come
 * in the Primary Server
 */

 private ServerSocket serverSoc;

/**
 * inStream is a DataInputStream Class object. It reads the messages from
 * the PrimaryServer which is written by OutputStream.
 */

 private DataInputStream inStream;

/**
 * port is an integer port number which server listens for connection.
 */

 private int port = 8888;

/**
 * allowedMiss is an integer value. It represents allowed number of packets
 * to be missed before deciding on Primary Server's failure.
 */

 private int allowedMiss;

/**
 * primaryServerStatus is a boolean value. If it is true, then it means that
 * Primary Server is up and running. If it is false it means that Primary
 * Server is down.
 */

 private boolean primaryServerStatus;

/**
 * backupLabel is a Label object. It is used for displaying "BACKUP SERVER"
 * text on the window.
 */

 private Label backupLabel;

/**
 * statusLabel is a Label object. It is used for displaying "STATUS" text
 * on the window.
 */
 private Label statusLabel;
```

```
/**
 * exitButton is a Button object. When clicked, exits the program.
 */

 private Button exitButton;

/**
 * statusDisplayLabel is a Label object. It is used for displaying the
 * status of the Primary Server. When the Primary Server is up and running,
 * it has a green background color and a text "NORMAL" on it. When the
 * Primary Server is down, it has a red background color and a text "PRIMARY
 * SERVER IS DOWN" on it.
 */

 private Label statusDisplayLabel;

/**
 * allowedMissLabel is a Label object. It is used for displaying "ALLOWED
 * NUMBER OF MISSES BEFORE FAILURE" text on the window.
 */

 private Label allowedMissLabel;

/**
 * allowedMissChoice is a Choice object. It is used for selecting the
 * allowedMiss value from drop-down list.
 */

 private Choice allowedMissChoice;

/**
 * intervalValue is a double value that is used for the heartbeat message
 * interval time in seconds.
 */

 private double intervalValue;

/**
 * lastMessageTime is a long value that is used for storing the last
 * received heartbeat message time in milliseconds.
 */

 private long lastMessageTime;

/**
 * inStr is a String that is used for holding the entire received message.
 */

 private String inStr;

/**
 * mainStr is a String that is used for holding the second part of the
 * received message.
 */
```

```
 private String mainString;

/**
 * initDelay is an integer value used for timer initialization
 */

 private int initDelay;

/**
 * Constructor for the BackupServer class. It initializes and sets the
 * specifications of the data members.
 *
 * @param none
 */

 public BackupServer() {

    try {

       //initialize the GUI components
       jbInit();
       //set default value for the allowed number of missed messages as 1
       allowedMiss = 1;
       lastMessageTime = 0;
       initDelay = 0;
       //initially assume that the Primary Server is up
       primaryServerStatus = true;
       //create the TimerHandler class and send this class as a reference
       timerHandler = new TimerHandler(this);

       try{
          //create the server socket which will listen on port 8888
          serverSoc = new ServerSocket(port);

       }
       catch( IOException ioe ){

          display.append("Could not listen on port " + port +"");
          display.append(ioe.getMessage());

       }//end try/catch

    }
    catch(Exception e) {

       e.printStackTrace();
       display.append(e.getMessage());

    }//end try/catch

 }//end BackupServer()

/**
 * This method is the main method of the BackupServer class. It creates a
```

```
 * BackupServer object and calls its start() method.
 *
 * @param args String[]
 * @return void
 */

public static void main(String[] args) {

   BackupServer backupServer1 = new BackupServer();
   backupServer1.start();

}//end main()


/**
 * This method initializes GUI components of the BackupServer class.
 *
 * @param none
 * @return void
 * @exception Exception
 */

private void jbInit() throws Exception {

   //create GUI components
   display = new TextArea();
   backupLabel = new Label();
   statusLabel = new Label();
   exitButton = new Button();
   statusDisplayLabel = new Label();
   allowedMissLabel = new Label();
   allowedMissChoice = new Choice();

   //initialize backupLabel
   backupLabel.setBackground(Color.lightGray);
   backupLabel.setBounds(new Rectangle(16, 25, 476, 52));
   backupLabel.setFont(new java.awt.Font("SansSerif", 1, 34));
   backupLabel.setForeground(Color.black);
   backupLabel.setAlignment(1);
   backupLabel.setText("BACKUP SERVER");

   //initialize statusLabel
   statusLabel.setBackground(Color.lightGray);
   statusLabel.setBounds(new Rectangle(15, 447, 170, 36));
   statusLabel.setFont(new java.awt.Font("Dialog", 1, 20));
   statusLabel.setAlignment(1);
   statusLabel.setText("STATUS");

   //initialize statusDisplayLabel
   statusDisplayLabel.setText("NORMAL");
   statusDisplayLabel.setAlignment(1);
   statusDisplayLabel.setFont(new java.awt.Font("Dialog", 1, 20));
   statusDisplayLabel.setBackground(new java.awt.Color(88, 255, 159));
   statusDisplayLabel.setBounds(new Rectangle(182, 447, 313, 35));
```

```
//initialize allowedMissLabel
allowedMissLabel.setText("ALLOWED NUMBER OF MISSES BEFORE FAILURE");
allowedMissLabel.setAlignment(1);
allowedMissLabel.setForeground(Color.black);
allowedMissLabel.setFont(new java.awt.Font("Dialog", 1, 15));
allowedMissLabel.setBackground(Color.lightGray);
allowedMissLabel.setBounds(new Rectangle(7, 502, 447, 24));

//initialize allowedMissChoice
allowedMissChoice.addItemListener(new java.awt.event.ItemListener() {
   public void itemStateChanged(ItemEvent e) {
      allowedMissChoice_itemStateChanged(e);
   }
});
allowedMissChoice.setBounds(new Rectangle(453, 500, 44, 24));
allowedMissChoice.setFont(new java.awt.Font("Dialog", 1, 15));

//add number from "1" through "4" to dropdown menu
allowedMissChoice.add("1");
allowedMissChoice.add("2");
allowedMissChoice.add("3");
allowedMissChoice.add("4");

//initialize exitButton
exitButton.setBackground(Color.lightGray);
exitButton.setBounds(new Rectangle(217, 549, 98, 35));
exitButton.setFont(new java.awt.Font("Dialog", 1, 20));
exitButton.setLabel("EXIT");
exitButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      exitButton_actionPerformed(e);
   }
});

//initialize display (Text Area)
display.setFont(new java.awt.Font("Dialog", 1, 14));
display.setBounds(new Rectangle(16, 78, 479, 366));
display.setBackground(Color.white);
display.setEditable(false);
display.append("\n\n" +
 "       Waiting for \"HEARTBEAT\" messages from Primary Server..\n");

//initialize frame properties
this.setLayout(null);
this.setBackground(Color.lightGray);
this.setTitle("Constant Heartbeat Protocol (V.0)");
this.addWindowListener(new java.awt.event.WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      this_windowClosing(e);
   }
});
this.add(exitButton, null);
this.add(display, null);
this.add(backupLabel, null);
this.add(allowedMissLabel, null);
```

```
    this.add(statusLabel, null);
    this.add(statusDisplayLabel, null);
    this.add(allowedMissChoice, null);
    this.setSize(515,600);
    this.setVisible(true);

 }//end jbInit()


/**
 * This method continuously listens for messages via ServerSocket. Whenever
 * a connection is made, it creates a new socket and retrieves the input
 * stream. This received stream consists of two parts. The first part has
 * the time interval information for the heartbeat messages. The second part
 * has heartbeat message. These two parts separated with the "~" delimiter.
 * When the first message received BackupServer does not know the heartbeat
 * interval. However Backup Server needs to know the heartbeat interval for
 * the correct initialization of the timer. Therefore, when message first
 * arrives it tokenizes the message and retrieves the heartbeat interval
 * value. Then, it creates the Timer class object called timer. Timer is
 * initialized to ((allowedMiss+1)*intervalValue). Whenever a heartbeat
 * message is received timer is restarted with the initial delay. Thus, as
 * long as the primary server continue to send the heartbeat messages, timer
 * never expires. The expiration of the timer indicates the failure of the
 * primary server
 *
 * @param none
 * @return void
 */

 private void start(){

    //this boolean value is for reading only the first message
    boolean oneTime = true;

    //loop if Primary Server is running
    while(primaryServerStatus){

       //call garbage collector
       System.gc();

       try{

          //when message received create a Socket
          Socket inSocket = serverSoc.accept();

          //disable drop-down menu
          allowedMissChoice.setEnabled(false);

          //retreive the stream from the socket
          inStream = new DataInputStream(inSocket.getInputStream());
          inStr = inStream.readLine();

          //tokenize the received inStr and set the intervalValue
          //and mainString data member values.
```

```
            tokenizeReceivedString();

            //read only the first message received and get the intervalValue
            //to be able to set the timer correctly
            //this block will be executed only once
            if (oneTime){

               //create the timer and set the initial delay
               initDelay=(int)(CONV_MILLIS*(allowedMiss+1)*(intervalValue));
               timer = new Timer(initDelay, timerHandler);
               timer.setRepeats(false);

               //set oneTime boolean as false. So do not enter in this block
               //again
               oneTime = false;

            }//end if

            //store the last message reception time
            lastMessageTime = System.currentTimeMillis();

            //reset the Timer value
            timer.setDelay(initDelay);

            //if timer is not running start it
            if(!timer.isRunning()){

               timer.start();

            }//end if

            //display message info
            display.append("\nReceived...:    " + "\" " +mainString+" \"");
            display.append("   Interval :  " + intervalValue );
            display.append("   at  :" + lastMessageTime + "");

         }
         catch (Exception ex){

           ex.printStackTrace();
           display.append( "\n\nException in creating thread\n");
           display.append(ex.getMessage());

         }//end try/catch

      }//end while

   }//end start()

  /**
   * If exitButton is clicked, terminates the program.
   *
   * @param e ActionEvent
   * @return void
   */
```

```
 private void exitButton_actionPerformed(ActionEvent e) {

    System.exit(1);

 }//end exitButton_actionPerformed()


/**
 * If "X" is clicked on window, terminates the program .
 *
 * @param e WindowEvent
 * @return void
 */

 private void this_windowClosing(WindowEvent e) {

    System.exit(1);

 }//end this_windowClosing()


/**
 * When a new value is selected for the allowedMiss, sets the allowedMiss
 * value
 *
 * @param e ItemEvent
 * @return void
 */

 private void allowedMissChoice_itemStateChanged(ItemEvent e) {

    allowedMiss = Integer.parseInt((e.getItem().toString()));

 }//end allowedMissChoice_itemStateChanged()



/**
 * Sets the status boolean to the parameter value. When status is set to
 * false displays the failure detection time. It also changes the color and
 * the text of the statusDisplayLabel.
 *
 * @param inStatus is a boolean for status
 * @return void
 */

 public void setPrimaryServerStatus(boolean inStatus){

    primaryServerStatus = inStatus;

    if (!primaryServerStatus){

       display.append("\n\nPRIMARY SERVER FAILED !!!!    AT :    "
                      + System.currentTimeMillis() + ""   );
```

```
            statusDisplayLabel.setBackground(new java.awt.Color(255, 0, 0));
            statusDisplayLabel.setText("PRIMARY SERVER IS DOWN");

            long detectionTime = System.currentTimeMillis();
            display.append("\n\nFailure detected at                    :  " +
                           detectionTime + "\n" +
                           "Last message received at          :  " +
                           lastMessageTime + "\n" +
                           "Elapsed time for detection is     :  " +
                           (detectionTime-lastMessageTime) + "  milliseconds");
         }//end if

      }//end setPrimaryServerStatus()


     /**
      * This method divides the received string into tokens with the delimiter
      * "~" and sets the intervalValue and the mainString data members
      *
      * @param none
      * @return void
      */

      private void tokenizeReceivedString(){

         //divide the string into tokens with delimiter "~"
         StringTokenizer tokens = new StringTokenizer(inStr,"~");
         int tokenNumber = tokens.countTokens();
         String [] tokenArray = new String[tokenNumber];

         for ( int i =0 ; i<tokenNumber ;i++){

            tokenArray[i] = tokens.nextToken();

         }//end for

         //first token has the intervalValue information
         intervalValue =Double.parseDouble(tokenArray[0]) ;

         //second token has the "I am alive" message
         mainString = tokenArray[1];

      }///end tokenizeReceivedString()

}//end class BackupServer

//end file BackupServer.java
```

```
//-----------------------------------------------------------------------------
// Filename:      TimerHandler.java
// Date     :     October 23, 1999
// Name     :     Efraim KATI
// Project :      Constant Heartbeat Protocol (V.0)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package constantheartbeat;

import java.awt.event.*;

/**
 * This class implements ActionListener and listens the action generated by
 * the timer of the BackupServer. When the timer expires, the actionPerformed
 * method of this class is called and the failure of the Primary Server is
 * declared.
 *
 * @author Efraim KATI
 */

public class TimerHandler implements ActionListener {

  /**
   * backupServer is a reference to a BackupServer object
   */

   BackupServer backupServer;

  /**
   * Constructor of this class
   */

   public TimerHandler(BackupServer inBackupServer) {

      backupServer = inBackupServer;

   }//end TimerHandler()

  /**
   * When the timer of the BackupServer is expires, this method declares the
   * failure of the Primary Server
   */

   public void actionPerformed(ActionEvent e) {

     backupServer.timer.stop();
     backupServer.setPrimaryServerStatus(false);

   }//end actionPerformed()

}//end TimerHandler class

//end file TimerHandler.java
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. THE ACCELERATED HEARTBEAT PROTOCOL SOURCE FILES

```
//-----------------------------------------------------------------------------
// Filename:       BackupServer.java
// Date     :      October 23, 1999
// Name     :      Efraim KATI
// Project :       Accelerated Heartbeat Protocol (V.1)
// Compiler:       JDK 1.2.2
//-----------------------------------------------------------------------------

package acceleratedheartbeat;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

/**
 * This class represents the Backup Server side of the "Accelerated
 * Heartbeat Protocol". In this prototype implementation, Backup Server
 * periodically emits "Are you alive?" messages (Heartbeat Queries) to the
 * Primary Server and listens the responses coming from the Primary Server.
 * The interval value for sending heartbeat queries is selected by the user
 * from the GUI. Message sending process is implemented with a repeating
 * timer. Whenever a timer expires a new Heartbeat Query message is sent.
 * The algorithm of the Accelerated Heartbeat Protocol is as follows:
 *
 *    1. If in a period, backup server sends a "Are you alive?" message to
 *       the primary server and receives a "I'm alive" message, then
 *       the backup server makes the length of the next period a large value
 *       "tmax" (irrespective on the length of the current period).
 *    2. If in a period, the backup server sends a "Are you alive?" message
 *       to the primary server but does not receive a "I'm alive" message,
 *       then the backup server makes the length of the next period half of
 *       the current period.
 *    3. If the length of the next period ever becomes less than a specified
 *       value "tmin", that is an upper bound on the round trip delay between
 *       backup server and primary server, than backup server declares the
 *       failure of the primary server.
 *
 * @author Efraim KATI
 */

public class BackupServer extends JFrame {

  /**
   * receiveServerSocket is a ServerSocket object and used for receiving
   * messages from the primary server
   */
```

```
 private ServerSocket receiveServerSocket;

/**
 * socket is Socket object and used for sending messages to the primary
 * server
 */

 private Socket socket;

/**
 * messageSendTimer is Timer object and used for sending messages
 */

 private Timer messageSendTimer;

/**
 * ackReceiveTimer is Timer object and used for message receive check
 */

 private Timer ackReceiveTimer;

/**
 * ackTimerHandler is AckTimerHandler object. It handles the actions
 * generated by the ackReceiveTimer
 */

 private AckTimerHandler ackTimerHandler;

/**
 * messageTimerHandler is MessageTimerHandler object. It handles the actions
 * generated by the messageSendTimer
 */

 private MessageTimerHandler messageTimerHandler;

/**
 * CONVERTER is a constant value used for converting seconds into
 * milliseconds
 */

 private final double CONVERTER = 1000.0;

/**
 * tmax is a double value that is used for the maximum time interval for
 * sending Heartbeat Query Messages
 */

 private double tmax;

/**
 * tmin is a double value that is used for the upper bound of the round trip
 * delay between servers
 */

 private double tmin;
```

```
/**
 * currentMessageSendTime is used for the last message transmit time
 */

 public long currentMessageSendTime;

/**
 * lastMessageReceiveTime is used for remembering the last message received
 * time
 */

 private long lastMessageReceiveTime;

/**
 * previousTime is used for remembering the previous message
 * transmit time
 */

 public long previousTime;

/**
 * rtdMax is used for calculating the maximum round trip delay
 */

 private long rtdMax;

/**
 * sendPort is the port number "8888" for sending messages
 */

 private int sendPort;

/**
 * receivePort is the port number "8889" for receiving messages
 */

 private int receivePort;

/**
 * lineCount is used for counting the displayed lines on the text area.
 * When the lineCount reaches 200, the text area is cleared. (Because
 * TextArea has some limitations)
 */

 private int lineCount;

/**
 * status is a boolean. If it is true then means that primary server
 * responding properly. If it is false then means that primary server
 * failed to respond
 */

 private boolean status;
```

```
/**
 * isPrimaryDown is a boolean. If it is true then means that primary server
 * is running. If it is false then means that primary server failed
 */

 private boolean isPrimaryDown;

/**
 * outMessage is the string that is sent to the primary server ("Are you
 * alive?")
 */

 private String outMessage;

/**
 * backupServerLabel is a JLabel object that displays the "BACKUP SERVER"
 * text on the GUI window
 */

 private JLabel backupServerLabel;

/**
 * display is a TextArea object that displays all outputs of the program on
 * the GUI window
 */

 private TextArea display;

/**
 * intervalLabel is a JLabel object that displays the ""SELECT "tmax" VALUE
 * FOR SENDING MESSAGES (sec) " text on the GUI window
 */

 private JLabel intervalLabel;

/**
 * tmaxComboBox is a JComboBox object that is used for selecting the "tmax"
 * value from a drop-down list
 */

 private JComboBox tmaxComboBox;

/**
 * startButton is a JButton object. When it is pressed, program starts
 * sending heartbeat query messages to the primary server
 */

 private JButton startButton;

/**
 * exitButton is a JButton object. When it is pressed, program is terminated
 */

 private JButton exitButton;
```

```
/**
 * intervalLabe2 is a JLabel object that displays the ""SELECT ROUND
 * TRIP DELAY UPPER BOUND (sec)" text on the GUI window
 */

 private JLabel intervalLabel2;

/**
 * roundTripDelayComboBox is a JComboBox object that is used for selecting
 * the "tmin" value from a drop-down list
 */

 private JComboBox roundTripDelayComboBox;

/**
 * exitButton is a JButton object. When it is pressed, program calculates
 * the upper bound of the round trip delay between servers
 */

 private JButton rtdButton;

/**
 * rtdDisplayLabel is a JLabel object that displays the calculated upper
 * bound of the round trip delay between servers
 */

 private JLabel rtdDisplayLabel;



/**
 * Constructor of the BackupServer class and initializes the data members
 * of this class
 * @param none
 */
 public BackupServer() {

    try  {

       jbInit();
       messageTimerHandler = new MessageTimerHandler(this);
       sendPort = 8888;
       receivePort = 8889;
       lineCount = 2;
       receiveServerSocket = new ServerSocket(receivePort);
       tmax = 0.5;
       tmin = 0.05;
       rtdMax = 0;
       status = true;
       isPrimaryDown=false;
       outMessage = "ARE YOU ALIVE?";

    }
    catch(Exception e) {
```

```
        display.append("\n" + e.getMessage());

    }//end try/catch

 }//end BackupServer()

/**
 * This method is the main method of the BackupServer class. It creates a
 * BackupServer object and calls its start() method.
 *
 * @param args String[]
 * @return void
 */

 public static void main(String[] args) {

    BackupServer backupServer21 = new BackupServer();

 }//end main()

/**
 * This method initializes GUI components of the BackupServer class.
 *
 * @param none
 * @return void
 * @exception Exception
 */

 private void jbInit() throws Exception {

    //creation GUI components
    backupServerLabel = new JLabel();
    display = new TextArea();
    intervalLabel = new JLabel();
    tmaxComboBox = new JComboBox();
    startButton = new JButton();
    exitButton = new JButton();
    intervalLabel2 = new JLabel();
    roundTripDelayComboBox = new JComboBox();
    rtdButton = new JButton();
    rtdDisplayLabel = new JLabel();

    //initialization of the backupServerLabel
    backupServerLabel.setFont(new java.awt.Font("Dialog", 1, 40));
    backupServerLabel.setBorder(BorderFactory.createRaisedBevelBorder());
    backupServerLabel.setHorizontalAlignment(SwingConstants.CENTER);
    backupServerLabel.setText("BACKUP SERVER");
    backupServerLabel.setBounds(new Rectangle(18, 3, 476, 53));

    //initialization of the display
    display.setBackground(new java.awt.Color(192, 231, 255));
    display.setBounds(new Rectangle(19, 61, 475, 315));
    display.setEditable(false);
    display.append("\n\n                                    INSTRUCTIONS");
    display.append("\n       ****************************************"+
```

```
                "****************************");
display.append("\n\n                    1. GET THE ROUND TRIP DELAY ");
display.append("\n\n                    2. SELECT THE \"tmax\" VALUE");
display.append("\n\n                    3. SET THE ROUND TRIP DELAY "+
                "VALUE ACCORDING TO ");
display.append("\n                      OBTAINED VALUE FROM STEP 1");
display.append("\n\n                    4. PRESS START BUTTON");

//initialization of the intervalLabel
intervalLabel.setFont(new java.awt.Font("Dialog", 1, 14));
intervalLabel.setText("SELECT \"tmax\" VALUE  FOR SENDING MESSAGES   "+
                      " (sec)");
intervalLabel.setBounds(new Rectangle(17, 417, 416, 29));

//initialization of the tmaxComboBox
tmaxComboBox.setBounds(new Rectangle(437, 419, 58, 23));
tmaxComboBox.addItem("0.5");
tmaxComboBox.addItem("1.0");
tmaxComboBox.addItem("1.5");
tmaxComboBox.addItem("2.0");
tmaxComboBox.addItem("2.5");
tmaxComboBox.addItem("3.0");
tmaxComboBox.addItemListener(new java.awt.event.ItemListener() {
   public void itemStateChanged(ItemEvent e) {
      tmaxComboBox_itemStateChanged(e);
   }
});

//initialization of the startButton
startButton.setFont(new java.awt.Font("Dialog", 1, 12));
startButton.setBorder(BorderFactory.createRaisedBevelBorder());
startButton.setText("START");
startButton.setBounds(new Rectangle(194, 486, 127, 28));
startButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      startButton_actionPerformed(e);
   }
});

//initialization of the exitButton
exitButton.setFont(new java.awt.Font("Dialog", 1, 12));
exitButton.setBorder(BorderFactory.createRaisedBevelBorder());
exitButton.setText("EXIT");
exitButton.setBounds(new Rectangle(209, 534, 97, 27));
exitButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      exitButton_actionPerformed(e);
   }
});

//initialization of the intervalLabel2
intervalLabel2.setBounds(new Rectangle(28, 403, 363, 29));
intervalLabel2.setBounds(new Rectangle(17, 436, 416, 29));
intervalLabel2.setText("SELECT ROUND TRIP DELAY UPPER BOUND        "+
                       "       (sec)");
```

```
intervalLabel2.setFont(new java.awt.Font("Dialog", 1, 14));
intervalLabel2.setBounds(new Rectangle(28, 403, 363, 29));
intervalLabel2.setBounds(new Rectangle(17, 448, 415, 29));

//initialization of the roundTripDelayComboBox
roundTripDelayComboBox.addItemListener(new java.awt.event.ItemListener(){
   public void itemStateChanged(ItemEvent e) {
      roundTripDelayComboBox_itemStateChanged(e);
   }
});
roundTripDelayComboBox.setBounds(new Rectangle(435, 405, 58, 23));
roundTripDelayComboBox.setBounds(new Rectangle(436, 439, 58, 23));
roundTripDelayComboBox.setBounds(new Rectangle(435, 405, 58, 23));
roundTripDelayComboBox.setBounds(new Rectangle(438, 449, 58, 23));
roundTripDelayComboBox.addItem("0.05");
roundTripDelayComboBox.addItem("0.06");
roundTripDelayComboBox.addItem("0.07");
roundTripDelayComboBox.addItem("0.08");
roundTripDelayComboBox.addItem("0.09");
roundTripDelayComboBox.addItem("0.10");
roundTripDelayComboBox.addItem("0.11");
roundTripDelayComboBox.addItem("0.12");
roundTripDelayComboBox.addItem("0.13");
roundTripDelayComboBox.addItem("0.14");
roundTripDelayComboBox.addItem("0.15");
roundTripDelayComboBox.addItem("0.16");
roundTripDelayComboBox.addItem("0.17");
roundTripDelayComboBox.addItem("0.18");
roundTripDelayComboBox.addItem("0.19");
roundTripDelayComboBox.addItem("0.20");
roundTripDelayComboBox.addItem("0.21");
roundTripDelayComboBox.addItem("0.22");
roundTripDelayComboBox.addItem("0.23");
roundTripDelayComboBox.addItem("0.24");
roundTripDelayComboBox.addItem("0.25");
roundTripDelayComboBox.addItem("0.26");
roundTripDelayComboBox.addItem("0.27");
roundTripDelayComboBox.addItem("0.28");
roundTripDelayComboBox.addItem("0.29");
roundTripDelayComboBox.addItem("0.30");

//initialization of the rtdButton
rtdButton.setFont(new java.awt.Font("Dialog", 1, 12));
rtdButton.setBorder(BorderFactory.createRaisedBevelBorder());
rtdButton.setText("GET ROUND TRIP DELAY");
rtdButton.setBounds(new Rectangle(18, 384, 181, 27));
rtdButton.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(ActionEvent e) {
      rtdButton_actionPerformed(e);
   }
});

//initialization of the rtdDisplayLabel
rtdDisplayLabel.setForeground(Color.red);
rtdDisplayLabel.setBounds(new Rectangle(208, 385, 287, 26));
```

```
    //initialization of the frame
    this.setTitle("Accelerated Heartbeat Protocol (V.1)");
    this.getContentPane().setLayout(null);
    this.getContentPane().add(intervalLabel, null);
    this.getContentPane().add(tmaxComboBox, null);
    this.getContentPane().add(roundTripDelayComboBox, null);
    this.getContentPane().add(exitButton, null);
    this.getContentPane().add(startButton, null);
    this.getContentPane().add(intervalLabel2, null);
    this.getContentPane().add(backupServerLabel, null);
    this.getContentPane().add(rtdButton, null);
    this.getContentPane().add(rtdDisplayLabel, null);
    this.getContentPane().add(display, null);
    this.setSize(520,600);
    this.setVisible(true);

 }//end jbInit()

/**
 * This method sends messages to the Primary Server. While primary server
 * responds properly the ackReceiveTimer is started with the delay of
 * (tmax/2) seconds. When the primary server fails to respond,
 * ackReceiveTimer is started with the half of the current delay. After the
 * message is sent, this messages starts the ackReceiveThread to receive
 * the response of the primary server
 *
 * @param none
 * @return void
 */

 public void sendMessage(){

    if (status){

       ackReceiveTimer.setInitialDelay((int)((tmax/2.0)*CONVERTER));
       ackReceiveTimer.restart();

    }
    else{

       ackReceiveTimer.setInitialDelay(ackTimerHandler.getNewDelay());
       ackReceiveTimer.restart();

    }//end if

    try{

       socket = new Socket(InetAddress.getLocalHost(), sendPort);
       socket.setTcpNoDelay(true);
       OutputStream out =socket.getOutputStream();
       StringBuffer outBuffer = new StringBuffer();
       outBuffer.append(outMessage);
       String outString = outBuffer.toString();
       out.write(outString.getBytes());
```

```
        out.flush();
        currentMessageSendTime = System.currentTimeMillis();
        long displayTime = currentMessageSendTime - previousTime;

        if (previousTime == 0){

           displayTime = 0;

        }//end if

        show("\n Sent Query at:  " + currentMessageSendTime +" diff :  "+
              displayTime + "");
        previousTime = currentMessageSendTime;
        AckReceiveThread receiveThread = new AckReceiveThread(this,
                                                  receiveServerSocket);
        receiveThread.start();
        System.gc();
        socket.close();

      }
      catch( Exception cex ){

        stopMessageSendTimer();

      }//end try/catch block

  }//end sendMessage()

 /**
  * This method calculates the upper bound of the round trip delay between
  * two servers, by sending four consecutive heartbeat query messages
  *
  * @param none
  * @return void
  */

  private void evaluateRTD(){

     long [] rtdArray = new long[4];
     long startTime = 0;
     long endTime = 0;

     try{

       for ( int i = 0 ; i < 4 ; i++){

           Socket rtdSocket = new Socket(InetAddress.getLocalHost(),
                                         sendPort);
           rtdSocket.setTcpNoDelay(true);
           OutputStream rtdOut =rtdSocket.getOutputStream();
           StringBuffer outBuffer = new StringBuffer();
           outBuffer.append("RTD TEST");
           String outString = outBuffer.toString();
           rtdOut.write(outString.getBytes());
           rtdOut.flush();
```

```
        startTime = System.currentTimeMillis();
        display.setText("");
        lineCount = 0;
        show("\n Testing Round Trip Delay.....");
        rtdOut.close();
        rtdSocket.close();
        Socket testSocket = receiveServerSocket.accept();
        endTime = System.currentTimeMillis();
        InputStream inStr = testSocket.getInputStream();
        DataInputStream inStream= new DataInputStream (inStr);
        String testString = inStream.readLine();
        inStream.close();
        testSocket.close();
        rtdArray[i] = endTime - startTime;
        System.gc();

    }//end for
}
catch( Exception ex ){

    show("\n" + ex.getMessage());

}//end try/catch block

long total =0;
for (int i=0 ; i<4 ; i++){

    total =+ rtdArray[i];

}//end for

long average = total/(long)4.0;
long tempMax1 = 0;
long tempMax2 = 0;

if(rtdArray[0] > rtdArray[1]){
    tempMax1 = rtdArray[0];
}
else{
    tempMax1 = rtdArray[1];
}
if(rtdArray[2] > rtdArray[3]){
    tempMax2 = rtdArray[2];
}
else{
    tempMax2 = rtdArray[3];
}
if(tempMax1 > tempMax2){
    rtdMax = tempMax1;
}
else{
    rtdMax = tempMax2;
}

show("\n Average Round-Trip-Delay is : " + average + " ms.");
```

```
    rtdDisplayLabel.setText("Max of four Round-Trip-Delay is : " + rtdMax+
                            " ms.");

 }//end evaluateRTD()

/**
 * When startButton is pressed, disables startButton, tmaxComboBox, and
 * rtdButton. Then starts the messageSendTimer. After that creates the
 * ackReceiveTimer. Finally clears the text area
 *
 * @param e ActionEvent
 * @return void
 */

 private void startButton_actionPerformed(ActionEvent e) {

    startButton.setEnabled(false);
    tmaxComboBox.setEnabled(false);
    roundTripDelayComboBox.setEnabled(false);
    rtdButton.setEnabled(false);
    messageSendTimer = new Timer((int)(tmax*CONVERTER),
                                        messageTimerHandler);
    messageSendTimer.start();
    ackTimerHandler = new AckTimerHandler(this);
    ackReceiveTimer = new Timer((int)((tmax/2.0)*CONVERTER),
                                        ackTimerHandler);
    display.setText("");
    lineCount = 0;

 }//end startButton_actionPerformed()

/**
 * If exitButton is clicked, terminates the program.
 *
 * @param e ActionEvent
 * @return void
 */

 private void exitButton_actionPerformed(ActionEvent e) {

    System.exit(1);

 }//end startButton_actionPerformed()

/**
 * When a new value is selected from the tmaxComboBox, sets the tmax value
 *
 * @param e ItemEvent
 * @return void
 */

 private void tmaxComboBox_itemStateChanged(ItemEvent e) {

    tmax = Double.parseDouble(e.getItem().toString());
```

```
 }//end tmaxComboBox_itemStateChanged()

/**
 * When a new value is selected from the roundTripDelayComboBox, sets the
 * tmin value
 *
 * @param e ItemEvent
 * @return void
 */

 private void roundTripDelayComboBox_itemStateChanged(ItemEvent e) {

    tmin =  Double.parseDouble(e.getItem().toString());


 }//end roundTripDelayComboBox_itemStateChanged()

/**
 * When a rtdButton is pressed this method is called. Then this method call
 * evaluateRTD method.
 *
 * @param e ActionEvent
 * @return void
 */

 private void rtdButton_actionPerformed(ActionEvent e) {

    evaluateRTD();

 }//end rtdButton_actionPerformed()

/**
 * Returns the tmax value
 *
 * @param none
 * @return tmax
 */

 public double getTmax(){

   return tmax;

 }//end getTmax()

/**
 * Returns the tmin value
 *
 * @param none
 * @return tmin
 */

 public double getTmin(){

   return tmin;
```

```
 }//end getTmin()

/**
 * Displays the passed string on the text area
 *
 * @param inString
 * @return void
 */

 public void show(String inString){

    display.append(inString);

    if(lineCount > 200){

       lineCount = 0;
       display.setText("");

    }//end if

    lineCount++;

 }//end show()

/**
 * Returns the status boolean
 *
 * @param none
 * @return status
 */

 public boolean getStatus(){

   return status;

 }//end getStatus()

/**
 * Sets the status boolean to a passed parameter value
 *
 * @param inStatus
 * @return void
 */

 public void setStatus(boolean inStatus){

   status = inStatus;

 }//end setStatus()

/**
 * Restarts the messageSendTimer
 *
 * @param none
 * @return void
```

```
 */

 public void restartMessageSendTimer(){

    messageSendTimer.restart();

 }//end restartMessageSendTimer()

/**
 * Stops the messageSendTimer
 *
 * @param none
 * @return void
 */

 public void stopMessageSendTimer(){

    messageSendTimer.stop();

 }//end stopMessageSendTimer()

/**
 * Stops the ackReceiveTimer
 *
 * @param none
 * @return void
 */

 public void stopAckReceiveTimer(){

    ackReceiveTimer.stop();

 }//end stopAckReceiveTimer()

/**
 * Returns the isPrimaryDown boolean
 *
 * @param none
 * @return isPrimaryDown
 */

 public boolean isPrimaryDown(){

    return isPrimaryDown;

 }//end isPrimaryDown()


/**
 * Sets the isPrimaryDown boolean to a false value
 *
 * @param none
 * @return void
 */
```

```
    public void setPrimaryDown(){

       isPrimaryDown = true;

    }//end setPrimaryDown()



   /**
    * Sets the lastMessageReceiveTime value to the passed parameter
    *
    * @param time
    * @return void
    */

    public void setLastMessageReceiveTime(long time){

      lastMessageReceiveTime = time;

    }//end setLastMessageReceiveTime()

   /**
    * Returns the lastMessageReceiveTime value
    *
    * @param none
    * @return lastMessageReceiveTime
    */

    public long getLastMessageReceiveTime(){

      return lastMessageReceiveTime;

    }//end getLastMessageReceiveTime()


}//end BackupServer class


//end file BackupServer.java
```

```
//-----------------------------------------------------------------------------
// Filename:      AckReceiveThread.java
// Date    :      October 23, 1999
// Name    :      Efraim KATI
// Project :      Accelerated Heartbeat Protocol (V.1)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package acceleratedheartbeat;

import javax.swing.*;
import java.net.*;
import java.io.*;
import java.awt.*;

/**
 * This class listens "I AM ALIVE" messages from the Primary Server. When
 * a message is received, it restarts the messageSendTimer if previously
 * stopped and stops the ackReceiveTimer.
 *
 * @author Efraim KATI
 */

public class AckReceiveThread extends Thread {

  /**
   * backupServer is a reference to a BackupServer object
   */

   private BackupServer backupServer;

  /**
   * receiveSocket is a reference to a ServerSocket object
   */

   private ServerSocket receiveSocket;

  /**
   * receivedStr is a String and used for the message received from the
   * Primary Server
   */

   private String receivedStr;

  /**
   * inStream is a DataInputStream object
   */

   private DataInputStream inStream;

  /**
   * receiveTime is a long value used for the receive time of the message
   */

   private long receiveTime;
```

```
/**
 * socket is a Socket object
 */

 private Socket socket;


/**
 * Constructor of this class
 * @param inBackupServer a BackupServer object
 * @param socket a Socket object
 */

 public AckReceiveThread(BackupServer inBackupServer,ServerSocket socket){

    backupServer = inBackupServer;
    receiveSocket = socket;

 }//end AckReceiveThread()

/**
 * This method receives the message from the Primary Server and stops the
 * ackReceiveTimer
 *
 * @param none
 * @return void
 */

 public void run(){

    if(!backupServer.isPrimaryDown()){

       try{

          socket = receiveSocket.accept();
          receiveTime = System.currentTimeMillis();

          if(!backupServer.getStatus()){

             backupServer.restartMessageSendTimer();
             backupServer.setStatus(true);

          }//end if

          backupServer.stopAckReceiveTimer();
          inStream = new DataInputStream(socket.getInputStream());
          receivedStr = inStream.readLine();
          backupServer.show("  Received response at : " +

                            receiveTime +"");

          socket.close();

       }
```

```
         catch( IOException ioe ){
            backupServer.show("\nCould not listen   " + ioe.getMessage());
         }//end try/catch block
      }//end if
   }//end run()
}//end AckReceiveThread class
//end file AckReceiveThread.java
```

```
//---------------------------------------------------------------------------
// Filename:      AckTimerHandler.java
// Date    :      October 23, 1999
// Name    :      Efraim KATI
// Project :      Accelerated Heartbeat Protocol (V.1)
// Compiler:      JDK 1.2.2
//---------------------------------------------------------------------------

package acceleratedheartbeat;

import java.awt.event.*;
import javax.swing.*;

/**
 * This class listens the actions generated by the ackReceiveTimer. Whenever
 * an action is performed it reduces the interval value by half. When the
 * interval becomes less than the upper bound of round trip delay(tmin),
 * it declares the failure of the Primary Server
 *
 * @author Efraim KATI
 */

public class AckTimerHandler implements ActionListener {

  /**
   * backupServer is a reference to a BackupServer object
   */

   private BackupServer backupServer;

  /**
   * lastMessageTime is a long value, used for holding the last message
   * sent-time
   */

   private long lastMessageTime;

  /**
   * newAckInterval is a double value, used as a delay value for the
   * ackReceiveTimer
   */

   private double newAckInterval = 0;

  /**
   * CONVERTER is a constant value, used for converting seconds to
   * milliseconds
   */

   private final double CONVERTER = 1000.0;


  /**
   * Constructor of this class
   * @param inBackupServer a BackupServer object
```

```
 */

 public AckTimerHandler(BackupServer inBackupServer) {

    backupServer = inBackupServer;

 }//end AckTimerHandler()


/**
 * Whenever the ackReceiveTimer expires this method is called.
 *
 * @param e ActionEvent
 * @return void
 */

 public void actionPerformed(ActionEvent e) {

    backupServer.show("     Did not receive response......");
    backupServer.show("\nAction performed ---------------------------->"
                      + System.currentTimeMillis()+"");

    if (backupServer.getStatus()){

       lastMessageTime = backupServer.currentMessageSendTime;
       newAckInterval = backupServer.getTmax()*CONVERTER/4.0;
       backupServer.setStatus(false);

    }
    else{

       newAckInterval = newAckInterval/2.0;

    }//end if

    if ((backupServer.getTmin()*CONVERTER) < newAckInterval ){

       backupServer.stopMessageSendTimer();
       backupServer.sendMessage();

    }
    else {// failure is detected

       long detectionTime = System.currentTimeMillis();
       backupServer.setPrimaryDown();
       backupServer.show("\n\nLast message send at  :" +
                         lastMessageTime +"");
       backupServer.show("\nFailure detected at   :" +
                         detectionTime +"");
       backupServer.show("\n\nElapsed Time          :" +
                        (detectionTime - lastMessageTime) +"");
       backupServer.show("\n\n*******PRIMARY SERVER IS DOWN ********");
       backupServer.stopMessageSendTimer();
       backupServer.stopAckReceiveTimer();
```

```
      }//end if

   }//end actionPerformed()


  /**
   * Returns an integer value to be set to the ackReceiveTimer delay
   */

   public int getNewDelay(){

      return ((int)newAckInterval);

   }//end getNewDelay()

}//end AckTimerHandler class

//end file AckTimerHandler.java
```

```
//-----------------------------------------------------------------------------
// Filename:      MessageTimerHandler.java
// Date     :     October 23, 1999
// Name     :     Efraim KATI
// Project  :     Accelerated Heartbeat Protocol (V.1)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package acceleratedheartbeat;

import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.net.*;

/**
 * This class implements ActionListener and listens the action generated by
 * the messageSendTimer of the BackupServer. When the timer expires, the
 * actionPerformed method of this class is called and a new "ARE YOU ALIVE?"
 * message is sent to the Primary Server
 *
 * @author Efraim KATI
 */

public class MessageTimerHandler implements ActionListener{

  /**
   * backupServer is a reference to a BackupServer object
   */

   private BackupServer backupServer;

  /**
   * Constructor of this class
   * @param inBackupServer a BackupServer object
   */

   public MessageTimerHandler(BackupServer inBackupServer) {

      backupServer = inBackupServer;

   }//end MessageTimerHandler()

  /**
   * When the messageSendTimer expires, this method is called. Then this
   * method calls the sendMessage method of the BackupServer class
   */

   public void actionPerformed(ActionEvent e) {

      backupServer.sendMessage();

   }//end actionPerformed()
```

```
}//end MessageTimerHandler class

//end file MessageTimerHandler.java
```

```
//---------------------------------------------------------------------------
// Filename:       PrimaryServer.java
// Date     :      October 23, 1999
// Name     :      Efraim KATI
// Project :       Accelerated Heartbeat Protocol (V.1)
// Compiler:       JDK 1.2.2
//---------------------------------------------------------------------------

package acceleratedheartbeat;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

/**
 * This class represents the Primary Server side of the "Accelerated
 * Heartbeat Protocol". In this class primary server listens the heartbeat
 * query messages from the backup server with a server socket. When a message
 * is received, a thread called PrimaryServerThread is created to handle the
 * connection and to send a respond to the backup server
 *
 * @author Efraim KATI
 */

public class PrimaryServer extends JFrame {

  /**
   * server1 is a ServerSocket object and used for receiving  messages from
   * the backup server
   */

   private ServerSocket server1;

  /**
   * port is the port number "8888" for receiving messages
   */

   private int port;

  /**
   * lineCount is used for counting the displayed lines on the text area. When
   * the lineCount reaches 200, the text area is cleared. (Because TextArea
   * has some limitations)
   */

   private int lineCount;

  /**
   * status is a boolean. If it is true then means that primary server is
   * running. If it is false then means that primary server failed
   */
```

```
 private boolean status;

/**
 * send is a boolean. If it is true, then response messages will be send.
 * If it is false, then response messages will not be send.
 */

 private boolean send;

/**
 * allowedMissLabel is a JLabel object. It is used for displaying "PRIMARY
 * SERVER" text on the window.
 */

 private JLabel primaryServerLabel;

/**
 * toggleButton is a JToggleButton object. It is used for either stopping
 * or starting the heartbeat response message transmission
 */

 private JToggleButton toggleButton;

/**
 * exitButton is a JButton object. When it is pressed, program is terminated
 */

 private JButton exitButton;

/**
 * display is a TextArea object that displays all outputs of the program on
 * the GUI window
 */

 public  TextArea display;

/**
 * Constructor of the PrimaryServer class. It initializes the data members
 * of this class
 * @param none
 */

 public PrimaryServer() {

    try  {
       //initialize GUI components
       jbInit();
       //initialize other data members
       status = true;
       send = true;
       port  = 8888;
       lineCount = 0;
       server1 = new ServerSocket(port);

    }
```

```
    catch( IOException ioe ){

          display.append("\nCould not listen   " +  ioe.getMessage());

    }//end catch
    catch(Exception e) {

       e.printStackTrace();
       display.append(e.getMessage());

    }//end try/catch block

 }//end PrimaryServer()

/**
 * This method is the main method of the PrimaryServer class. It creates a
 * PrimaryServer object and calls its start() method.
 *
 * @param args String[]
 * @return void
 */

 public static void main(String[] args) {

    PrimaryServer primaryServer = new PrimaryServer();
    primaryServer.start();

 }//end main()

/**
 * This method initializes GUI components of the PrimaryServer class.
 *
 * @param none
 * @return void
 * @exception Exception
 */

 private void jbInit() throws Exception {

    //creation GUI components
    primaryServerLabel = new JLabel();
    toggleButton = new JToggleButton();
    exitButton = new JButton();
    display = new TextArea();

    //initialization of the primaryServerLabel
    primaryServerLabel.setBackground(new java.awt.Color(255, 159, 175));
    primaryServerLabel.setFont(new java.awt.Font("Dialog", 1, 40));
    primaryServerLabel.setBorder(BorderFactory.createRaisedBevelBorder());
    primaryServerLabel.setToolTipText("");
    primaryServerLabel.setDisplayedMnemonic('0');
    primaryServerLabel.setHorizontalAlignment(SwingConstants.CENTER);
    primaryServerLabel.setText("PRIMARY SERVER");
    primaryServerLabel.setBounds(new Rectangle(15, 11, 450, 44));
```

```
    //initialization of the toggleButton
    toggleButton.setText("STOP SENDING RESPONSES");
    toggleButton.setFont(new java.awt.Font("Dialog", 1, 12));
    toggleButton.setBounds(new Rectangle(103, 407, 277, 36));
    toggleButton.addActionListener(new java.awt.event.ActionListener() {
       public void actionPerformed(ActionEvent e) {
          toggleButton_actionPerformed(e);
       }
    });

    //initialization of the exitButton
    exitButton.setBorder(BorderFactory.createRaisedBevelBorder());
    exitButton.setText("EXIT");
    exitButton.setFont(new java.awt.Font("Dialog", 1, 12));
    exitButton.setBounds(new Rectangle(193, 480, 96, 31));
    exitButton.addActionListener(new java.awt.event.ActionListener() {
       public void actionPerformed(ActionEvent e) {
          exitButton_actionPerformed(e);
       }
    });

    //initialization of the display
    display.setBackground(new java.awt.Color(255, 231, 183));
    display.setBounds(new Rectangle(16, 67, 449, 327));
    display.setText("WAITING FOR \"ARE YOU ALIVE?\""+
                    " MESSAGES FROM BACKUP SERVER");
    display.setEditable(false);

    //initialization of the frame
    this.getContentPane().setLayout(null);
    this.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    this.setEnabled(true);
    this.setTitle("Accelerated Heartbeat Protocol (V.1)");
    this.addWindowListener(new java.awt.event.WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          this_windowClosing(e);
       }
    });
    this.getContentPane().add(primaryServerLabel, null);
    this.getContentPane().add(exitButton, null);
    this.getContentPane().add(toggleButton, null);
    this.getContentPane().add(display, null);
    this.setSize(500,600);
    this.setVisible(true);

 }//end jbInit()

/**
 * This method listens on port 8888 with server socket. When a connection
 * is made a new PrimaryServerThread is created.
 *
 * @param none
 * @return void
 */
```

```
private void start(){

   while(status){

      try{

         Socket inSocket = server1.accept();
         inSocket.setTcpNoDelay(true);
         PrimaryServerThread primaryServerThread = new PrimaryServerThread(
                                                   this,
                                                   inSocket);
         primaryServerThread.start();

      }
      catch (Exception ex){

         status = false;
         ex.printStackTrace();
         display.append(ex.getMessage());

      }//end try/catch block

      System.gc();

   }//end while

}//end start()

/**
 * When the toggleButton is pressed this method is called. If currently
 * primary server is sending responses, then it stops message transmission.
 * If currently is not sending, then it restarts sending responses
 *
 * @param e ActionEvent
 * @return void
 */

private void toggleButton_actionPerformed(ActionEvent e) {

   if (send){

      send = false;
      toggleButton.setBackground(Color.white);
      toggleButton.setText("SEND RESPONSES AGAIN");

   }
   else{

      send = true;
      toggleButton.setBackground(new java.awt.Color(151, 167, 247));
      toggleButton.setText("STOP SENDING RESPONSES");

   }//end if

}//end toggleButton_actionPerformed()
```

```
/**
 * If exitButton is clicked, terminates the program.
 *
 * @param e ActionEvent
 * @return void
 */

 private void exitButton_actionPerformed(ActionEvent e) {

    System.exit(0);

 }//end exitButton_actionPerformed()

/**
 * If "X" is clicked on window, terminates the program .
 *
 * @param e WindowEvent
 * @return void
 */

 private void this_windowClosing(WindowEvent e) {

    System.exit(1);

 }//end this_windowClosing()

/**
 * Returns the send boolean
 *
 * @param none
 * @return send
 */

 public boolean isSend(){

    return send;

 }//end isSend()

/**
 * Increments the lineCount value with the passed parameter. When the
 * lineCount value exceeds 200, it is set to zero and the text area is
 * cleared
 *
 * @param increase a value to be added to the lineCount
 * @return void
 */

 public void incrementLineCount(int increase){

    if (lineCount > 200){

       lineCount = 0;
       display.setText("");
```

```
      }//end if
      lineCount = lineCount + increase;
   }//end incrementLineCount()
}//end PrimaryServer class
//end file PrimaryServer.java
```

```
//-----------------------------------------------------------------------------
// Filename:      PrimaryServerThread.java
// Date    :      October 23, 1999
// Name    :      Efraim KATI
// Project :      Accelerated Heartbeat Protocol (V.1)
// Compiler:      JDK 1.2.2
//-----------------------------------------------------------------------------

package acceleratedheartbeat;

import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * This class is created by the PrimaryServer class. The purpose of this class
 * is to handle the established connection with the Backup Server, by receiving
 * the message and sending the response. (if the isSend boolean of the
 * PrimaryServer is true)
 *
 * @author Efraim KATI
 */

public class PrimaryServerThread extends Thread{

  /**
   * display is a reference to the TextArea object of the PrimaryServer
   */

   private TextArea display;

  /**
   * primaryServer is a reference to the PrimaryServer object
   */

   private PrimaryServer primaryServer;

  /**
   * socket is a reference to the socket of the PrimaryServer
   */

   private Socket socket;

  /**
   * inStr is a String that is used for the received message
   */

   private String inStr;

  /**
   * outStr is a String that is used for the outgoing message
   */

   private String outStr;
```

```
/**
 * Constructor of the PrimaryServerThread class. It initializes the data
 * members of this class
 *
 * @param none
 */

 public PrimaryServerThread( PrimaryServer pServer, Socket inSocket) {

    primaryServer = pServer;
    display = primaryServer.display;
    socket = inSocket;
    inStr = null;
    outStr = "YES. I AM ALIVE";

 }//end PrimaryServerThread()

/**
 * Run method of this thread. When start() is called this method is executed.
 * This method receives the incoming message and sends the outgoing message
 *
 * @param none
 * @return void
 */

 public void run(){

    try{

       DataInputStream inStream= new DataInputStream(
                                          socket.getInputStream());
       inStr = inStream.readLine();
       display.append("\nReceived Query at: "+System.currentTimeMillis());
       primaryServer.incrementLineCount(1);

       if (primaryServer.isSend()){

          try{

             Socket sendSocket = new Socket(InetAddress.getLocalHost(),
                                            8889);
             OutputStream outStream = sendSocket.getOutputStream();
             StringBuffer ackBuffer = new StringBuffer();
             ackBuffer.append(outStr);
             String outString = ackBuffer.toString();
             outStream.write(outString.getBytes());
             outStream.flush();
             display.append("   Sent response at: " +
                           System.currentTimeMillis() + "");

             outStream.close();
             socket.close();

          }
```

```
            catch(IOException e){

               display.append("\n" + e.getMessage());

            }//end try/catch
          }
         else{

            display.append("    RESPONSE DID NOT SEND ");

            try{

               inStream.close();
               socket.close();

            }
            catch(IOException e){

               display.append("\n" + e.getMessage());

            }//end try/catch

         }//end if

      }
      catch (Exception ex){

         display.append("\n" + ex.getMessage());

      }//end try/catch

   }// end run()

}//end PrimaryServerThread class

//end file PrimaryServerThread.java
```

APPENDIX C. NEWLY ADDED SOURCE FILES FOR INTEGRATION

```
//---------------------------------------------------------------------------
// Filename : HeartbeatQuery.java
// Date     : December 8, 1999
// Author   : Efraim KATI
// Project  : SAAM
//---------------------------------------------------------------------------

package saam.message;

import saam.util.*;
import java.io.*;

/**
 * This Class will be used by the backup server to query the main server
 * status. HeartbeatQuery message contains a one-byte type field (type
 * equals to three), and a two-byte sequence number field.
 *
 * @author Efraim KATI
 */

public class HeartbeatQuery extends Message{

  /**
   * bytes is a byte array object that stores the byte code representation of
   * this class.
   */

   private byte[] bytes = null;

  /**
   * sequenceNumber is a short value used to determine if the HeartbeatResponse
   * is the response for the last sent HeartbeatQuery message.
   */

   private short sequenceNumber;

  /**
   * counter is static short value that is incremented by one everytime this
   * object is created and used for assigning the value of the sequenceNumber
   */

   private static short counter = 0;

  /**
   * Parameterless constructor. Constructs a HeartbeatQuery message.
   * @param none
   */

   public HeartbeatQuery(){
```

```
        super(Message.HEARTBEAT_QUERY_TYPE);
        counter++;
        sequenceNumber = counter;
        bytes = Array.concat(bytes,
                             PrimitiveConversions.getBytes(sequenceNumber));

    }//end HeartbeatQuery()

    /**
     * Constructs a HeartbeatQuery message with the byte representation of
     * this class.
     * @param bytes byte array representation of this class.
     */

    public HeartbeatQuery(byte[] bytes) {

        super(Message.HEARTBEAT_QUERY_TYPE);
        this.bytes = null;
        this.bytes = bytes;
        sequenceNumber = PrimitiveConversions.getShort(
                         Array.getSubArray(bytes,0, bytes.length));

    }//end HeartbeatQuery()

    /**
     * Returns The byte array representation of this Message.
     * @param none
     * @return The byte array representation of this Message.
     */

    public byte[] getBytes(){

        return bytes;

    }//end getBytes()

    /**
     * Returns the sequenceNumber.
     * @param none
     * @return sequenceNumber
     */

    public short getSequenceNumber(){

        return sequenceNumber;

    }//end getSequenceNumber()

    /**
     * Returns the length of this Message.
     * @param none
     * @return The length of this Message.
     */

    public short length(){
```

```
      try{

         return (short)bytes.length;

      }catch(NullPointerException npe){

         return 0;

      }//end try-catch block

   }//end length()

  /**
   * Returns a String representation of this Message.
   * @param none
   * @return The String representation of this Message
   */

   public String toString(){

      return ("\n\"HeartbeatQuery Message\"" +
              "\nTYPE = " + type +
              "\nSequence Number = " + sequenceNumber + "");

   }//end toString()

}//end class HeartbeatQuery

//end file HeartbeatQuery.java
```

```
//---------------------------------------------------------------------------
// Filename : HeartbeatResponse.java
// Date     : December 8, 1999
// Author   : Efraim KATI
// Project  : SAAM
//---------------------------------------------------------------------------

package saam.message;

import saam.util.*;
import java.io.*;

/**
 * This Class is used by Main server as a response to the HeartbeatQuery
 * message. HeartbeatQuery message contains a one-byte type field (type equals
 * to two) and a two-byte sequence number field.
 *
 * @author Efraim KATI
 */

public class HeartbeatResponse extends Message{

  /**
   * bytes is a byte array object that stores the byte code representation of
   * this class.
   */

   private byte[] bytes;

  /**
   * sequenceNumber is a short value used to determine if the HeartbeatResponse
   * is the response for the last sent HeartbeatQuery message.
   */

   private short sequenceNumber;

  /**
   * lastUsedFlowID is the value of the last assigned flowID by the main server
   */

   private int lastUsedFlowID;

  /**
   * Constructs a HeartbeatResponse message.
   * @param lastUsedFlowID integer value for last used flow ID.
   * @param sequenceNumber from the HeartbeatQuery Message which will be
   *        responded.
   */

   public HeartbeatResponse(short sequenceNumber, int lastUsedFlowID){

      super(Message.HEARTBEAT_RESPONSE_TYPE);

      this.sequenceNumber = sequenceNumber;
      this.lastUsedFlowID =  lastUsedFlowID;
```

```
    bytes=Array.concat(bytes,PrimitiveConversions.getBytes(sequenceNumber));
    bytes=Array.concat(bytes,PrimitiveConversions.getBytes(lastUsedFlowID));

 }//end HeartbeatResponse()

/**
 * Constructs a HeartbeatResponse message with the byte representation of
 * this class
 * @param bytes byte array representation of this class.
 */

 public HeartbeatResponse(byte[] bytes) {

    super(Message.HEARTBEAT_RESPONSE_TYPE);
    this.bytes = null;
    this.bytes = bytes;
    sequenceNumber = PrimitiveConversions.getShort(Array.getSubArray(bytes,
                                                                     0,
                                                                     2));
    lastUsedFlowID = PrimitiveConversions.getInt(Array.getSubArray(bytes,
                                                                   2,
                                                                   6));

 }//end HeartbeatResponse()

/**
 * Returns The byte array representation of this Message.
 * @param none
 * @return The byte array representation of this Message.
 */

 public byte[] getBytes(){

    return bytes;

 }//end getBytes()

/**
 * Returns The last used flow id value.
 * @param none
 * @return The last used flow_id value.
 */

 public int getLastUsedFlowID(){

    return lastUsedFlowID;

 }//end getLastUsedFlowID()

/**
 * Returns The sequenceNumber value.
 * @param none
 * @return The sequenceNumber.
 */
```

```
    public short getSequenceNumber(){

       return sequenceNumber;

    }//end getSequenceNumber()

   /**
    * Returns the length of this Message.
    * @param none
    * @return The length of this Message.
    */

    public short length(){

       try{

          return (short)bytes.length;

       }catch(NullPointerException npe){

          return 0;

       }//end try/catch

    }//end length()

   /**
    * Sets the lastUsedFlowID value
    * @param lastUsedFlowID The last used flow_ID value
    * @return void
    */

    public void setLastUsedFlowID(int lastUsedFlowID){

       this.lastUsedFlowID = lastUsedFlowID;
       bytes = null;
       bytes =
Array.concat(bytes,PrimitiveConversions.getBytes(lastUsedFlowID));

    }//end setLastUsedFlowID()


   /**
    * Sets the sequenceNumber
    * @param sequenceNumber The last used flow_ID value
    * @return void
    */

    public void setSequenceNumber(short sequenceNumber){

       this.sequenceNumber = sequenceNumber;
       bytes = null;
       bytes =
Array.concat(bytes,PrimitiveConversions.getBytes(sequenceNumber));
```

```
      bytes =
Array.concat(bytes,PrimitiveConversions.getBytes(lastUsedFlowID));

   }//end setSequenceNumber()

  /**
   * Returns a String representation of this Message.
   * @param none
   * @return The String representation of this Message.
   */

   public String toString(){

      return ("\n\"HeartbeatResponse Message\"" +
              "\nTYPE = " + type +
              "\nSequenceNumber = " + sequenceNumber +
              "\nLast Used Flow ID = " + lastUsedFlowID + "");

   }//end toString()


}//end class HeartbeatResponse

//end file HeartbeatResponse.java
```

```
//-----------------------------------------------------------------------------
// Filename : HeartbeatController.java
// Date     : December 8, 1999
// Author   : Efraim KATI
// Project  : SAAM
//-----------------------------------------------------------------------------

package saam.server;

import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import saam.util.*;

/**
 * The HeartbeatController class is responsible for periodically sending the
 * heartbeat query messages and performing a check on the time constraints of
 * the heartbeat response messages.
 *
 * In order to periodically send the heartbeat query messages, the
 * HeartbeatController class uses a Timer called querySendTimer. The initial
 * delay of the querySendTimer is set to "tmax". The querySendTimer is a
 * repeating timer. Whenever the querySendTimer expires, the actionPerformed
 * method of the QuerySendTimerHandler class (inner class of the
 * HeartbeatController class) is executed. From this actionPerformed method,
 * the sendHertbeatQuery method of the Server class is called.
 *
 * In order to perform a check on the time constraints of the heartbeat
 * response messages, the HeartbeatController class uses another Timer called
 * responseControlTimer. The responseControlTimer's initial delay is set to the
 * half of the  value. Whenever a correct heartbeat response message is
 * received from the primary server, the responseControlTimer is stopped, and
 * whenever a new heartbeat query message is sent it is restarted with the half
 * of the current interval time. Therefore, as long as the heartbeat response
 * messages are received properly the responseControlTimer never expires. If
 * the backup server sends a heartbeat query message, but does not receive a
 * heartbeat response message within the first half of the current period, then
 * the responseControlTimer expires.
 *
 * Whenever the responseControlTimer expires, the actionPerformed method of the
 * responseControlTimerHandler class, (an inner class of the
 * HeartbeatController class), is executed. In this actionPerformed method, the
 * sendHertbeatQuery method of the Server class is called, and the length of
 * the next interval period is reduced by half. If the length of the next
 * interval period ever becomes less than the  , then the time of the last
 * received DCM message is checked. If a new DCM is received within the last
 * period, then a failure declaration is not declared. If a new DCM is not
 * received within the last  period, then failure declaration of the primary
 * server is declared by calling the setIsMainDown method of the Server class.
 *
 * @author Efraim KATI
 */

public class HeartbeatController {
```

```
/**
 * tmax is a double used for the time interval between the heartbeat query
 * messages in normal condition.
 */
 private double tmax;

/**
 * tmin is a double used for the round-trip delay upper bound between the two
 * servers.
 */
 private double tmin;

/**
 * querySendTimerHandler is a QuerySendTimerHandler class object used for
 * periodically sending the heartbeat query messages.
 */
 private QuerySendTimerHandler querySendTimerHandler;

/**
 * responseControlTimerHandler is a ResponseControlTimerHandler class object
 * used to perform a check on the time constraints of the heartbeat
 * response messages.
 */
 private ResponseControlTimerHandler responseControlTimerHandler;

/**
 * querySendTimer is a Timer class object. The querySendTimer's initial delay
 * is set to "tmax' value. The querySendTimer is a repeating timer. Whenever
 * the querySendTimer expires, the actionPerformed method of the
 * QuerySendTimerHandler class is executed.
 */
 private Timer querySendTimer;

/**
 * responseControlTimer is a Timer class object. The responseControlTimer's
 * initial delay is set to the half of the "tmax" value. Whenever a correct
 * heartbeat response message is received from the primary server, the
 * responseControlTimer is stopped, and whenever a new heartbeat query
 * message is sent it is restarted with the half of the current interval time
 */
 private Timer responseControlTimer;

/**
 * server is a reference to a Server class object.
 */
 private Server server;

/**
 * status is a boolean used for the status of the HeartbeatController class.
 * If status is true, then it means that the primary server is properly
 * responding the heartbeat query messages. If the status is false, then it
 * means that primary server is not failed yet but recently the backup server
 * is failed to receive one or more heartbeat response messages.
 */
 private boolean status;
```

```
/**
 * bf is a BannerFrame object used for displaying some heartbeat protocol
 * related messages on it.
 */
 private BannerFrame bf;

/**
 * counter is an integer used for counting the recently missed consecutive
 * heartbeat response messages.
 */
 private static int counter = 0;


/**
 * Constructor of the HeartbeatController class. It initializes data members
 * of this class
 * @param server is a Server class object
 * @param tmax is time interval between heartbeat query messages
 * @param tmin is the round-trip delay upper bound
 * @param bf is a BannerFrame class object
 */
 public HeartbeatController(Server server,
                            double tmax,
                            double tmin,
                            BannerFrame bf) {

    this.server = server;
    this.tmax = tmax;
    this.tmin = tmin;
    this.bf = bf;
    querySendTimerHandler = new QuerySendTimerHandler();
    responseControlTimerHandler = new ResponseControlTimerHandler();
    querySendTimer = new Timer((int)(tmax*1000.0d), querySendTimerHandler);
    responseControlTimer = new Timer((int)((tmax/2.0d)*1000.0d),
                                     responseControlTimerHandler);
    status = true;

 }//end HeartbeatController()

/**
 * Starts the querySendTimer.
 * @param none
 * @return void
 */
 public void startSending(){

    querySendTimer.start();

 }//end startSending()

/**
 * Sets the status boolean
 * @param inStaus a boolean to be set to the status boolean
 * @return void
```

```
 */
 public void setStatus(boolean inStatus){

    status = inStatus;
    if (status){

       bf.setBackgroundColor(Color.cyan);
       bf.setFrameText("THIS IS THE BACKUP SERVER");
       counter = 0;

    }//end if

 }//end setStatus()

/**
 * Returns the status boolean.
 * @param none
 * @return status
 */
 public boolean getStatus(){

    return status;

 }//end getStatus()

/**
 * Stops the responseControlTimer
 * @param none
 * @return void
 */
 public void stopResponseControlTimer(){

    responseControlTimer.stop();

 }//end stopResponseControlTimer()

/**
 * Restarts the responseControlTimer
 * @param none
 * @return void
 */
 public void restartResponseControlTimer(){

    if(status){

       responseControlTimer.setInitialDelay((int)((tmax/2.0d)*1000d));
       responseControlTimer.restart();

    }
    else{

       responseControlTimer.setInitialDelay(
                  (int)responseControlTimerHandler.getNewControlInterval());
       responseControlTimer.restart();
```

```
      }//end if

   }//end restartResponseControlTimer()

  /**
   * Restarts the querySendTimer()
   * @param none
   * @return void
   */
   public void restartQuerySendTimer(){

      querySendTimer.restart();

   }//end restartQuerySendTimer()



  /**
   * The ResponseControlTimerHandler class is an inner class of the
   * HeartbeatController class. This class is used for listening the actions
   * generated by the responseControlTimer.
   *
   * @author Efraim KATI
   */
   private class ResponseControlTimerHandler implements ActionListener{

     /**
      * newControlInterval is a double used for the next period's time
interval.
      */
      private double newControlInterval = 0.0d;

     /**
      * Whenever the responseControlTimer expires, this method is executed.
      * In this method, the sendHertbeatQuery method of the Server class is
      * called, and the length of the next interval period is reduced by half.
      * If the length of the next interval period ever becomes less than the
      * "tmin", then the time of the last received DCM message is checked.
      * If a new DCM is received within the last period, then a failure
      * declaration is not declared. If a new DCM is not received within the
      * last   period, then failure declaration of the primary server is
      * declared by calling the setIsMainDown method of the Server class.
      */
      public void actionPerformed(ActionEvent e) {

         server.display("ResponseControlTimer  >>>>>>>>>>>>>>  "+
                        "actionPerformed at : " + System.currentTimeMillis()+"");
         if (status){

            newControlInterval = tmax*1000d/4.0d;
            status =false;

         }
         else{
```

```
   newControlInterval = newControlInterval/2.0d;

}//end if


if ((newControlInterval*2.0d) > tmin*1000d){

   server.addRecentMissedSequences();
   counter++;
   //stop sending timer
   querySendTimer.stop();
   //send query message immediately
   server.sendHeartbeatQuery();
   bf.setBackgroundColor(Color.orange);
   bf.setFrameText("PRIMARY SERVER MISSED " + counter +" RESPONSE");
   bf.setVisible(true);
   server.display("************************************************"+
                  "************************************************");
}
else {

   long lastDCM = server.getLastDCMTime();
   long current = System.currentTimeMillis();
   long diff = current - lastDCM;

   if(diff < (tmax*1000.0d)){

      server.display("\n\nDidn't receive heartbeat response message "+
            "but DCM is still coming. So failure detection will be "+
            "delayed one more cycle ");

      if(!status){

         restartQuerySendTimer();
         status = true;

      }//end if

      stopResponseControlTimer();

   }
   else{

      querySendTimer.stop();
      responseControlTimer.stop();
      long lastMessageTime = server.getLastResponseTime();
      long detectionTime = System.currentTimeMillis();
      long elapsedTime = detectionTime - lastMessageTime ;
      server.setIsMainDown(true);
      server.display("\n\n******** Primary Server is down and I "+
                     "became the Primary Server *******");
      server.display("\n\nFailure Detected at : " +detectionTime+ "");
      server.display("\n\nLast Heartbeat Response is received at : "+
                     lastMessageTime + "");
      server.display("\n\nElapsed Time  : " + elapsedTime + "");
```

```
                bf.setBackgroundColor(Color.red);
                bf.setForegroundColor(Color.yellow);
                bf.setFrameText("PRIMARY SERVER IS DOWN !.....");
                server.gui.setTextField("The Backup Server is active now");

             }//end if

          }//end if

       }//end actionPerformed()

      /**
       * Returns the newControlInterval
       * @param none
       * @return newControlInterval
       */
       public double getNewControlInterval(){

          return newControlInterval;

       }//end getNewControlInterval()

    }//end ResponseControlTimerHandler class



   /**
    * The QuerySendTimerHandler class is an inner class of the
    * HeartbeatController class. This class is used for listening the actions
    * generated by the querySendTimer.
    *
    * @author Efraim KATI
    */
    private class QuerySendTimerHandler implements ActionListener {

      /**
       * Whenever the querySendTimer expires, this method is executed.
       * From this actionPerformed method, the sendHertbeatQuery method of the
       * Server class is called.
       */
       public void actionPerformed(ActionEvent e) {

          server.sendHeartbeatQuery();

       }//end actionPerformed()

    }//end QuerySendTimerHandler class

}//end HeartbeatController class

//end file HeartbeatController.java
```

```
//-----------------------------------------------------------------------------
// Filename : BannerFrame.java
// Date     : December 8, 1999
// Author   : Efraim KATI
// Project  : SAAM
//-----------------------------------------------------------------------------

package saam.util;

import java.awt.*;
import javax.swing.*;

/**
 * The BannerFrame class is a GUI component added to the current GUI of the
 * Server class to display some accelerated heartbeat protocol information to
 * the user. Specifically, the BanerFrame class is a Java class extended from
 * the JFrame. It has only one constructor that accepts a string for display.
 * Additionally, the BannerFrame class has three set methods. The setFrameText
 * method is used to change the currently displayed text. The
 * setBackgroundColor method is used to change the background color of the
 * frame, and the setForegroundColor is used to change the color of the
 * displayed text on the frame. Initially, according to the server type either
 * "THIS IS THE PRIMARY SERVER" text or "THIS IS THE BACKUP SERVER" text is
 * displayed on the bannerFrame. If the backup server fails to receive a
 * heartbeat response message, then "PRIMARY SERVER MISSED n RESPONSE" text is
 * displayed (n is the number of missed messages). Additionally, if the backup
 * server declares the failure of the primary server, then "PRIMARY SERVER IS
 * DOWN" text is displayed.
 *
 * @author Efraim Kati
 */
public class BannerFrame extends JFrame {

  /**
   * jLabel1 is a JLabel object. It is used for displaying some messages on the
   * frame.
   */
   JLabel jLabel1 = new JLabel();

  /**
   * Constructor for the BannerFrame class. It initializes and sets the
   * specifications of the data members of this class.
   * @param text to be displayed
   */
   public BannerFrame(String text) {

      try  {

         jbInit();
         jLabel1.setText(text);

      }
      catch(Exception e) {

         e.printStackTrace();
```

```
    }//end try/catch

 }//end BannerFrame()

/**
 * This method initializes GUI components of the BannerFrame class.
 * @param none
 * @return void
 * @exception Exception
 */
 private void jbInit() throws Exception {

    jLabel1.setBackground(Color.pink);
    jLabel1.setFont(new java.awt.Font("Dialog", 1, 40));
    jLabel1.setHorizontalAlignment(SwingConstants.CENTER);
    jLabel1.setHorizontalTextPosition(SwingConstants.CENTER);
    jLabel1.setText("");
    this.getContentPane().setBackground(Color.pink);
    this.getContentPane().add(jLabel1, BorderLayout.CENTER);
    Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    float widthFactor = 1.3f;
    float heigthFactor = 8.4f;
    setSize((int)(dim.width/(widthFactor)),(int)(dim.height/(heigthFactor)));
    setLocation((int)(dim.width/2)-(int)(dim.width/(widthFactor)/2), 0);

 }//end jbInit()

/**
 * Sets the displayed string.
 * @param str is a string to be displayed
 * @return void
 */
 public void setFrameText(String str){

    jLabel1.setText(str);

 }//end setFrameText()

/**
 * Sets the background color
 * @param c is a Color to be set
 * @return void
 */
 public void setBackgroundColor(Color c){

    this.getContentPane().setBackground(c);

 }//end setBackgroundColor()

/**
 * Sets the color of the displayed text
 * @param c is a Color to be set
 * @return void
 */
```

```
    public void setForegroundColor(Color c){
        jLabel1.setForeground(c);
    }//end setForeGroundColor()

}//end BannerFrame class
//end file BannerFrame.java
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. MODIFIED SOURCE FILES FOR INTEGRATION

In the following class files, modified sections are highlighted with light gray.

```
//---------------------------------------------------------------------------
// Filename    : Message.java
// Date        : December 8, 1999
// Author      : John YARGER
// Modified by : Efraim KATI
// Project     : SAAM
//---------------------------------------------------------------------------

package saam.message;


/**
 * The Message class provides a convenient way for Objects to communicate
 * with one another over a SAAM network.  The standard JDK does not currently
 * provide a means to serialize objects over UDP.  This class does just that.
 * Subclasses need to be written as follows to enable this functionality:
 *
 * 1. Provide a constructor that accepts a byte array as its only parameter.
 * 2. Override the getBytes method in such a way that it returns a byte array
 *    that contains the values of the variables to be transferred.
 * 3. Ensure that the constructor mentioned above is set up to properly
 *    parse the byte array and rebuild the variables as they were originally.
 * 4. Ensure that the length method returns the actual length of the byte
 *    array.
 */

public abstract class Message{

   //for default type to support old version
   public static final byte MESSAGE_DEFAULT_TYPE    = 1;

   //for fault tolerance
   public static final byte HEARTBEAT_QUERY_TYPE    = 2;
   public static final byte HEARTBEAT_RESPONSE_TYPE = 3;

   //for control channel construction
   public static final byte UCM_TYPE                = 4;
   public static final byte DCM_TYPE                = 5;
   public static final byte PARENT_NOTIFICATION_TYPE= 6;
   public static final byte RESERVED1_TYPE          = 7;

   //following types reserved for flow reservation
   public static final byte RESERVED2_TYPE          = 8;
   public static final byte RESERVED3_TYPE          = 9;
   public static final byte RESERVED4_TYPE          = 10;
   public static final byte RESERVED5_TYPE          = 11;
```

```
public static final byte RESERVED6_TYPE          = 12;

//following types reserved for probing
public static final byte RESERVED7_TYPE          = 13;
public static final byte RESERVED8_TYPE          = 14;
public static final byte RESERVED9_TYPE          = 15;

//following types reserved for resource management
public static final byte RESERVED10_TYPE         = 16;
public static final byte RESERVED11_TYPE         = 17;
public static final byte RESERVED12_TYPE         = 18;

//following types are reserved for security
public static final byte RESERVED13_TYPE         = 19;
public static final byte RESERVED14_TYPE         = 20;
public static final byte RESERVED15_TYPE         = 21;
public static final byte RESERVED16_TYPE         = 22;

/**
 * type is a byte value to represent different type of messages
 */

protected byte type;

/**
 * No-args constructor initializes the type to a default value which is 1.
 * @param none
 */

public Message(){

   type =  MESSAGE_DEFAULT_TYPE ;

}//end Message()

/**
 * Constructs a Messagee with the supplied type_id parameter.
 * @param type_id byte value representing different types of messages
 */

public Message(byte type_id){

   this.type = type_id;

}//end Message()


/**
 * Returns the type value.
 * @param none
 * @return The type value.
 */

public byte getType(){
```

```
        return type;

    }//end getType()


   /**
    * Abstract method. Returns the length of this Message.
    * @param none
    * @return The length of this Message.
    */

    public abstract short length();

   /**
    * Abstract method. Returns The byte array representation of this Message.
    * @param none
    * @return The byte array representation of this Message.
    */

    public abstract byte[] getBytes();

   /**
    * Returns a String representation of this Message.
    * @param none
    * @return The String representation of this Message
    */

    public String toString(){

        return "Message";

    }//end toString()

}//end class Message

//end Message.java
```

```
//-----------------------------------------------------------------------------
// Filename     : Server.java
// Date         : December 8, 1999
// Author       : John YARGER (August 1999)
// Modified by  : Hasan AKKOC (February 2000)
// Modified  Efraim KATI (February 2000)
// Project      : SAAM
//-----------------------------------------------------------------------------

package saam.server;

import saam.EmulationTable;
import saam.Translator;
import saam.*;
import saam.net.*;
import saam.message.*;
import saam.control.*;
import saam.event.*;
import saam.router.*;
import saam.util.*;

import java.net.*;
import java.util.*;
import java.io.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionEvent;


/**
 * The <em>Server</em> is an object within the SAAM architecture that
 * maintains a picture of the network for use in assigning flows to paths.
 */

public class Server implements  Runnable{

  //declare class variables

  /** Contains what is known about the network. */
  private PathInformationBase PIB;

  /** Enables the Server to receive and send particular types of messages. */
  public ControlExecutive controlExec;

  /** A maximum number of hops that a search for different paths may take. */
  private int Hmax = 4;

  /**
   * Used to lookup what flow id should be used to send out control messages
   * to specified routers.
   */
  private Hashtable flowLookUp = new Hashtable();

  /** Used to assign the right number of service level pipes to interfaces in
   * this SAAM region. Only used during initialization -- later were assume
```

```
 * SLPs are known to routers
 */
private int numOfServiceLevels = 4;

/**
 * The value assigned to flow ids that can not be supported. This should be
 * switched over to 0 as soon as routers are converted.
 */
public static int FLOWNOTSUPPORTABLE = 99;

public static int INITIALDELAY = 0;
public static int INITIALLOSSRATE = 0;
public static int INITIALTHROUGHPUT = 10000;

public static int RETURNFLOWDELAY = 50;
public static int RETURNFLOWLOSSRATE = 50;
public static int RETURNFLOWTHROUGHPUT = 1000;

public static int ROUTERNOTINPIB = 0;

public static int NOSUPPORTABLEPATHINPIB = 0;

public static int SERVERNODEID = 1;

public static int FLOWTOSERVER = 0;

public static int PSUEDORANDOMSOURCEPORT = 8000;

public static int INITIALPATHID = 0;

public static int INITIALHEIGHTOFSEARCH = 1;
public static int INCREMENTATIONOFSEARCH = 1;
public static int DESTINATIONNODE = 0;

public static int INITIALZERO = 0;


/** Defines with the appropriate IPv6 address of this server. */
//private String serverIPv6 = controlExec.getServerIP().toString();
private String serverIPv6 = "99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2";

/** Time when the all possible paths were found. */
private long timeOfLastPIBBuild = System.currentTimeMillis();

/**
 * The amount of time that we want to have between rebuilding of paths. This
 * is not currently implemented.
 */
private long timeBetweenPIBBuilds = 120000; // 2 minutes (or 120 sec)

/** A boolean that will allow the showing of comments. */
private boolean showComments = true;

public SAAMRouterGui gui;
```

```
  // 2000 akkoc added
  private int sequenceNumber = 1;
  private static final int CTS = 0;     //SINCE ITS SERVER BY ITSELF
  private int hopCount;


  private static byte serverType;
  private static int flowId;
  private static byte metricType;
  private static int cycleTime;
  private static int globalTime;

  //1 Feb 2000 akkoc added
  private IPv6Address ServerId;

  //(February 2000) the following data members are added by Efraim KATI
  private int lastUsedFlow_ID = 0;
  private short seqNumInit = 0;
  private HeartbeatResponse hbr = new HeartbeatResponse(seqNumInit,
                                                        lastUsedFlow_ID);
  private HeartbeatController hbController;
  private boolean isMain;
  private boolean isMainDown = false;
  private short lastSequenceNumber = 0;
  private Vector recentMissedSequences = new Vector();
  private double tmax = 80.0d;
  private double tmin = 2.5d;
  private BannerFrame bf = new BannerFrame("");
  private long lastResponseTime = System.currentTimeMillis();
  private long lastQueryTime = System.currentTimeMillis();
  private long currentTime = System.currentTimeMillis();
  private int firstQueryTime = 80000;
  private long lastDCMTime = System.currentTimeMillis();
  private String primaryServerIPv6Str = "99.99.99.99.0.0.0.0.0.0.0.0.0.0.1";
  private String backupServerIPv6Str = "99.99.99.99.8.0.0.0.0.0.0.0.0.0.2";

//*************************************************************************
//  THE FOLLOWING TEST CASES ARE NOT PART OF THE CODE. THEY ARE ADDED ONLY FOR
//  CREATING SEVERAL TEST CONDITIONS (Efraim KATI)
//*************************************************************************

  //*************
  // TEST CASE #1
  // this case tests normal failure detection capability of the implemented
  // accelerated heartbeat protocol. For this test case no boolean is necessary
  // because the normal operation of the protocol provides the necessary test
  // condition.
  //
  // private boolean testCase1 = false; //(not used)

  //*************
  // TEST CASE #2
  // this case tests the lost heartbeat response
  // after the 10th heartbeat query message (if testCase2 is true):
  // * 3th respone will not be send
```

```
// * 4th response will not be send
// then responses will be sent correctly
private boolean testCase2 = false;

//*************
// TEST CASE #3
// this case tests the sequence number functionality
// after the 10th heartbeat query message (if testCase3 is true):
// * 11th respone will not be send
// * 12th response will be send with sequence # 11
// then responses will be sent correctly
private boolean testCase3 = false;

//*************
// TEST CASE #4
// this case tests the usage of DCM as unsolicited heartbeat
// after the 10th heartbeat query message (if testCase4 is true):
// * 11th respone will not be send
// * 12th response will not be send
// * 13th respone will not be send
// * 14th response will not be send
// * 15th response will not be send
// then responses will be sent correctly
private boolean testCase4 = false;


/**
 * Constructs a server that will use a specified type of <em>Path Information
 * Base</em>.  The PIB may be in the form of a database structure (which
 * requires an existing ODBC configured local database) or a class
 * object structure. The control executive is the interface to the IPv6
 * protocol stack, in order for messages to flow to and from the network.
 * The final step taken is the deletion of all existing data, which is
 * important only in a database structure since a class object structure is
 * volatile.
 * @param type The type of structure that the PIB is to assume.
 * @param controlExec The control executive that will exchange messages
 * with this server.
 */
public Server(String type, ControlExecutive controlExec){
  if (type == "database")
    PIB = new DatabaseStructure();
  else
    PIB = new ClassObjectStructure();

  this.controlExec = controlExec;
  gui=new SAAMRouterGui("Server");

  // 1feb 2000 akkoc added
  ServerId = controlExec.getRouterId();

  PIB.deleteAllData();

}
```

```
  public Server(){}//temp for time measurement



//*****************************************************************************
// These methods handle external network communications from routers
//*****************************************************************************/


  /**
   * Receives Hello messages from routers and then processes them.  It starts
   * building a vector of IPv6Addresses from the interfaces included in the
   * Hello message. This vector is passed to the PIB's doesRouterExist() which
   * determines if a router with any of these interfaces have been identified
   * before. If this is a new router, a new unique node id is assigned.<p>
   * For each of the interfaces identified in the Hello message, if this
   * interface was is not known to the PIB, check to see if the corresponding
   * link is known to the PIB. If this link is not known to the PIB, add it.
   * Next, add the new interface between the node and link. Also, add each
   * service level pipe that is assigned within this SAAM region.<p>
   * The next step is to rebuild the paths that are possible across the network
   * now considering this new hello message. The frequency of these rebuilds is
   * not a major concern in a controlled environment, but will need to be
   * addressed later. Finally, a flow request is create and received for
   * communicating back to this node. This is only possible if the PIB's
   * determineAllPossiblePaths() has been executed after the processing of this
   * particular hello message, if this a new router. After all paths to each
   * known router are found, we finish this method with a call to
   * determineEffectiveQoSForPaths(). The call to
   * determineEffectiveQoSForPaths() ensures that even if no QoS parameters are
   * known about these new parts of the network, that at least some initial
   * values will be assigned. This initialization allows the new paths to be
   * assigned if needed.
   * @param hello An initialization message from a router.
   */
  public void processHello(Hello hello) {

    long start, finish;
    Vector interfaces;
    int node_id = INITIALZERO;
    InterfaceID myInterface;
    int bandwidth = INITIALZERO;
    IPv6Address address = new IPv6Address();
    Vector IPv6Addresses = new Vector();
    boolean newRouter = true;
    FlowRequest myFlowRequest = new FlowRequest();

    // capture the start time of processing a hello
    start = System.currentTimeMillis();

    // produce a vector of IPv6Addresses
    interfaces = hello.getInterfaceIDs();
    for (int i = INITIALZERO; i < interfaces.size(); i++){
      address = ((InterfaceID)interfaces.elementAt(i)).getIPv6();
```

```
      IPv6Addresses.addElement(address);
    }

    // check if router exists and if so, return it's node id, else return 0
    node_id = PIB.doesRouterExist(xist(IPv6Addresses);

    // if the router does not exist in PIB
    if (node_id == ROUTERNOTINPIB){
      // assign it a new node id
      node_id = PIB.getNewNodeId();
    } else {
      newRouter = false;
    }

    // run through all of the LSA interfaces
    for (int i = INITIALZERO; i < interfaces.size(); i++) {
      myInterface = (InterfaceID)interfaces.elementAt(i);
      address = myInterface.getIPv6();
      // if a new interface is not found in the PIB, then ...
      PIB.doesInterfaceExist(address)){
        bandwidth = myInterface.getBandwidth();
        address = myInterface.getIPv6();
        // if the link is not contained in the PIB, then add it
        if (!PIB.doesLinkExist(address)){
          PIB.addLink(address, bandwidth);
        }
        // now add the interface between the node and the link
        PIB.addInterface(node_id, address);
        // now add each service level pipe
        for (int service_level = 0; service_level < numOfServiceLevels;
                                                          service_level++){
          PIB.addSLP(address, service_level, INITIALDELAY, INITIALLOSSRATE,
            INITIALTHROUGHPUT);
        }
      } // end if
    } //end interfaces for

    // capture the hello processing finish time
    finish = System.currentTimeMillis();
      gui.sendText("Server: processHello: Time required = "
        +(finish-start)+" milliseconds.");

    // time since last PIB build is > 2 min and if node did not exist before
    //if ((timeOfLastPIBBuild - System.currentTimeMillis())
>timeBetweenPIBBuilds
    //                                                          &&
newRouter){

      // rebuild all possible paths
      findAllPossiblePaths();

      // determine effective QoS of each path
      determineEffectiveQoSForPaths();

      // construct a new flow to this router
```

```
    try{
      myFlowRequest = new FlowRequest(IPv6Address.getByName(serverIPv6),
        address,System.currentTimeMillis(),RETURNFLOWDELAY,
        RETURNFLOWLOSSRATE,RETURNFLOWTHROUGHPUT);
    } catch(UnknownHostException uhe){
      System.err.println("Server: main: UnknownHostException: " + uhe);
    }
    processFlowRequest(myFlowRequest);
  //}

} //end processFlowRequest


/**
 * Receives link state advertisement messages from router and processes the
 * service level pipe status information that they contain. It begins by
 * checking to see if a router with the interface address described by this
 * LSA is known to the PIB. If such a router is known to exist, it then
 * checks to see if the service level pipe described by this LSA is known to
 * the PIB. If the service level pipe is known, then update its status.
 * Otherwise, add the SLP with the specified QoS characteristics. Finally,
 * update the effective QoS for the paths that pass over this service level
 * pipe by calling the determineEffectiveQoSForPaths().
 * @param router A representation of a router as defined by an LSA.
 */
public void processLSA(LinkStateAdvertisement LSA) {

  long start, finish;
  int node_id = INITIALZERO;
  int bandwidth = INITIALZERO;
  byte service_level = 0;
  int delay = INITIALZERO;
  int loss_rate = INITIALZERO;
  int utilization = INITIALZERO;
  Vector interfaces = new Vector(3,1);
  Vector SLPs = new Vector(3,1);
  IPv6Address link_id;
  IPv6Address address;
  Vector IPv6Addresses = new Vector(1,1);

  // capture the start time of processing an LSA
  start = System.currentTimeMillis();

  // produce a one element vector of IPv6Addresses
  address = LSA.getMyIPv6();
  IPv6Addresses.addElement(address);

  // check if router exists and if so, return it's node id, else return 0
  node_id = PIB.doesRouterExist(IPv6Addresses);

  // if the router does exist in PIB, then so does the interface...
  if (node_id != ROUTERNOTINPIB){

    service_level = LSA.getServiceLevel();
    delay = LSA.getDelay();
```

```
    loss_rate = LSA.getLossRate();
    utilization = LSA.getUtilization();

    if (showComments){
      gui.sendText("Server: processLSA: node_id = " + node_id
        + ", address = " + address + ", SL = "+service_level
        +", D = " +delay+ ", LR = "+loss_rate
        +", U = "+utilization);
    }

    // if this SLP is defined, then just update its status
    if(PIB.doesSLPExist(address, service_level))
    {
      PIB.updateSLP(address, service_level, delay, loss_rate, utilization);
    }
    // otherwise, insert it
    else {
      PIB.addSLP(address, service_level, delay, loss_rate, utilization);
    } // end else
  } // end if
  else { //do nothing
  }

  // capture the LSA processing finish time
  finish = System.currentTimeMillis();
  gui.sendText("Server: processLSA: Time required = "
    +(finish-start)+" milliseconds.");

  // revise the effective QoS of paths made up of this SLP
  determineEffectiveQoSForPaths(address,service_level);

} //end processLSA


/**
 * Receives and processes flow requests from applications. It begins
 * by finding a source and a destination router. These routers may be where
 * the applications are residing themselves, which is our standard situation.
 * The application could, however, reside on some host that is not registered
 * with the PIB as a router. In this case, the appropriate source or
 * destination router would be a router connected to the same link. <p>
 * The PIB is checked to ensure that there is the effective QoS available on
 * some path to satisfy the request. If a satisfactory path is found, a new
 * unique flow id is assigned and this new flow is associated with that path.
 * Each router in the path is retrieved and a new flow routing table entry is
 * sent to each. If no path can provide the requested level of QoS, then the
 * flow is assigned to zero, which will be interpreted by IPv6 as best effort
 * traffic. Finally, a flow response is sent back to the application to
 * inform it of its assigned flow id. If the flow id that is return is zero,
 * it will be the application's responsibility to either lower it QoS request
 * or to send its traffic as best effort.
 * @param flow_request The message requesting the establishment of a flow.
 */
public void processFlowRequest(FlowRequest flow_request) {
```

```
/** A vector of slp_sequence information for a path. */
Vector slps_in_path;
SLPSequence currentSLPSequence,nextSLPSequence = new SLPSequence();
int SLP_source_router, SLP_destination_router, service_level;
IPv6Address link_id = new IPv6Address();
IPv6Address next_hop;
IPv6Address sourceAddress;
int source_router, destination_router, path_id,
  flow_id=FLOWNOTSUPPORTABLE;
long start, finish;

// capture the start time of processing a flow request
start = System.currentTimeMillis();

// find a router on the same subnet as the source host
source_router =
  PIB.findARouterOnLink(flow_request.getSourceInterface());

// find a router on the same subnet as the destination host
destination_router =
  (PIB.findARouterOnLink(flow_request.getDestinationInterface()));

path_id = PIB.getPathThatCanSupportFlowRequest(source_router,
                                          destination_router, flow_request);

// if a path can support this request, then...
if(path_id != NOSUPPORTABLEPATHINPIB){

  // assign a flow id to the request
  flow_id = PIB.getNewFlowId(path_id,source_router,destination_router,
                                                          flow_request);

  lastUsedFlow_ID = flow_id;

  // determine each router in path
  // transmit Flow Routing Table Entry to it
  slps_in_path = PIB.getSLPSequenceOfPath(path_id);
  // for each router in the path, send a FRTE update
  for (int index = INITIALZERO; index < slps_in_path.size(); index++){
    // assign new slp sequence object
    currentSLPSequence = (SLPSequence)slps_in_path.elementAt(index);

    // if not the last link..
    if (index+1 != slps_in_path.size()){
      nextSLPSequence = (SLPSequence)slps_in_path.elementAt(index+1);
    }

    // retrieve values from this object
    SLP_source_router = currentSLPSequence.getSourceRouter();
    link_id = currentSLPSequence.getLinkId();
    service_level = currentSLPSequence.getServiceLevel();

    // if not the last link...
    if (index+1 != slps_in_path.size()){
      SLP_destination_router = nextSLPSequence.getSourceRouter();
```

```
        } else {
          // else it is the destination node of the flow
          SLP_destination_router = destination_router;
        }
        // determine destination address for next hop
        next_hop = PIB.getInterfaceAddress(SLP_destination_router, link_id);

        // determine source address
        sourceAddress = PIB.getInterfaceAddress(SLP_source_router, link_id);

        // send the flow routing table entry update
        sendFRTEUpdate(sourceAddress, flow_id, next_hop, service_level);

      } // end for

    } // end if

    //give routers time to finish updating tables
    try{
      Thread.sleep(2000);
    }catch(InterruptedException ie){
      gui.sendText(ie.toString());
    }
     // if the source of this flow is the server,
    if (source_router == SERVERNODEID) {
      // then add this new flow to hash table for later lookup
      if (showComments){
        gui.sendText("Server: processFlowRequest: use flow "+flow_id
          +" to send to node "+destination_router);
      }
      if (destination_router == SERVERNODEID) {
        flow_id = FLOWTOSERVER;
      }
      flowLookUp.put(new Integer(destination_router),new Integer(flow_id));
    }

    sendFlowResponse(flow_request, flow_id);

    // capture the flow request processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: processFlowRequest: Time required = "
      +(finish-start)+" milliseconds.");
  }

  /**
   * Receives flow termination from routers and then processes them.
   */
  public void receiveFlowTermination() { }


//*****************************************************************************
// These methods handle external network communications to routers
//*****************************************************************************/

  /**
```

```
 * Sends a flow routing table entry update message to a router. This message
 * provides the router the required information to forward packets based on
 * its flow id.
 * @param sourceAddress The router that will receive the FRTE update.
 * @param flow_id The id assigned to the flow in question.
 * @param next_hop The IPv6 address of the next node in the path.
 * @param service_level The service level that this flow is assigned to.
 */
public void sendFRTEUpdate(IPv6Address sourceAddress, int flow_id,
                                    IPv6Address next_hop, int service_level) {
  if(showComments){
    gui.sendText("Server: sendFRTEUpdate: flowLookUp hashtable:");
    gui.sendText(""+flowLookUp);
  }
  FlowRoutingTableEntry myFRTE = new FlowRoutingTableEntry(flow_id,
                                                  (byte)service_level, next_hop);
  int sourcePort = PSUEDORANDOMSOURCEPORT;
                                          //controlExec.listenToRandomPort(this);
  short destPort = ControlExecutive.SAAM_CONTROL_PORT;
  IPv6Address destHost = sourceAddress;
  // take steps to determine what flow id to send the packet on
  Vector interfaces = new Vector();
  interfaces.addElement(destHost);
  int destNodeId = PIB.doesRouterExist(interfaces);
  int flowIdToSendItOn = ((Integer)flowLookUp.get
                                       (new Integer(destNodeId))).intValue();
  try{
    controlExec.send(this,myFRTE, flowIdToSendItOn, (short)sourcePort,
                                                          destHost, destPort);
  } catch (FlowException fe){
    System.err.println(fe.toString());
  }
  if (showComments){
    gui.sendText("Server: sendFRTEUpdate: FRTE for flow " + flow_id
      + " sent to interface "+sourceAddress);
    gui.sendText("          with next hop= "+next_hop
      +" on service level "+service_level+" via flow "+flowIdToSendItOn);
  }
}



/**
 * Sends a flow response to the requesting application to notify it of
 * its newly assigned flow id. A flow id of zero is used to indicate that the
 * flow cannot be supported. Once a flow response message is instantiated and
 * a source and destination port is defined, the control executive's send()
 * is called to send it to the destination host.
 * @param flow_request The flow request message that was received.
 * @param flow_id The flow id that is assigned to the flow request.
 */
public void sendFlowResponse(FlowRequest flow_request, int flow_id){
  if(showComments){
    gui.sendText("Server: sendFlowResponse: flowLookUp hashtable:");
    gui.sendText(""+flowLookUp);
```

```
    }
    FlowResponse response = new FlowResponse(flow_request.getTimeStamp(),
      flow_id);
    int sourcePort = PSUEDORANDOMSOURCEPORT;
                                        //controlExec.listenToRandomPort(this);
    short destPort = ControlExecutive.SAAM_CONTROL_PORT;
    IPv6Address destHost = flow_request.getSourceInterface();
    // take steps to determine what flow id to send the packet on
    Vector interfaces = new Vector();
    interfaces.addElement(destHost);
    int destNodeId = PIB.doesRouterExist(interfaces);
    int flowIdToSendItOn = ((Integer)flowLookUp.get
                                        (new Integer(destNodeId))).intValue();
    try{
      controlExec.send(this, response, flowIdToSendItOn, (short)sourcePort,
                                                          destHost, destPort);
    }catch(FlowException fe){
      System.err.println(fe.toString());
    }
    if (showComments){
      gui.sendText("Server: sendFlowResponse: Flow response "
        + response + " from SourcePort: "+sourcePort+" to "+destHost
        + " sent via flow "+flowIdToSendItOn);
    }

  }

//*****************************************************************************
// These methods handle internal manipulation of data describing network status
//*****************************************************************************/

  /**
   * Determines all of the possible paths that exist between any source and
   * destination router in the network. This determination is based on the
   * physical definition of the network that is provided by the hello messages
   * received from the routers and stored within the PIB. The paths that are
   * found are then recorded in the PIB for fast assignment of flows later.<p>
   * All node ids are first retrieved from the PIB. For each service level, we
   * build an array of parents of each node. A parent is node that is directly
   * connected. Those directly connected nodes would have service level pipes
   * that would need to be passed through to get to the child node in question.
   * This parent array is used to populate a path table. Each node id is
   * assigned as the final destination of path and all of the different paths
   * are then found by working out from this destination. For each of these
   * destination nodes, a call is made to processPath() to find all the valid
   * paths that go to this destination node. We make the call with a specified
   * height of search of 1.
   */
  public void findAllPossiblePaths() {
    long start, finish;
    int NumberOfRouters;
    int max_slp_id = INITIALZERO;
    /** A count of the highest path id assigned so far. */
    int max_path_id = INITIALZERO;
    int service_level = INITIALZERO;
```

```
    /** A vector of the routers that are known by the db. */
    Vector V = new Vector();

    /** A vector of the parent routers for each given destination router. */
    Hashtable parent;

    // capture the start time of processing a path data
    start = System.currentTimeMillis();

    // reset the maximum path id assigned so far to zero
    max_path_id = INITIALPATHID;

    V = PIB.getAllRouterIds();

    //retrieve COUNT of routers
    NumberOfRouters = V.size();

    //find all possible paths for each service level
    max_slp_id = (new Integer(PIB.findMaxServiceLevel())).intValue();
    for(service_level=INITIALZERO;service_level<=max_slp_id; service_level++){

      //build parent array of each SLP at this service level
      parent = PIB.getParents(V, service_level);

      //populate path table
      for (int index = INITIALZERO; index < NumberOfRouters; index++){
        int heightOfSearch = INITIALHEIGHTOFSEARCH;
        int aPath[] = new int[Hmax + INCREMENTATIONOFSEARCH];
        aPath[DESTINATIONNODE] = ((Integer)V.elementAt(index)).intValue();
        processPath(parent, aPath, heightOfSearch,service_level);
      }
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: findAllPossiblePaths: Time required = "
      +(finish-start)+" milliseconds.");

    timeOfLastPIBBuild = finish;

  }

  /**
   * Processes all valid paths that arrive at the destination node within some
   * range of hops. For each parent of the node at the distance of
   * heightOfSearch from the destination, a check is made to ensure that adding
   * this new parent will cause no cycle. If this checks out, then that parent
   * can be added and a new path can be assigned. The service level pipes in
   * this new path are identified and their sequence numbers in this path are
   * recorded to the PIB. Next, a check is made to see if the height of the
   * search is less than the server's max search height of Hmax. If it is less,
   * the method recursively calls itself with an incremented heightOfSearch
   * variable.
   * @param parent Contains each router and a list of other
```

```
 * routers that are directly attached to them.
 * @param aPath[] An array contain a path from a source node,
 * aPath[heightOfSearch], to a destination node, aPath[0].
 * @param heightOfSearch The number of nodes in the path so far.
 * @param service_level The level of service assigned to a flow.
 */

public void processPath(Hashtable parent,
                        int aPath[], int heightOfSearch, int service_level){
  IPv6Address link_id;
  int justARouter;
  int sequence_number;
  int path_id;
  Enumeration W = ((Vector)parent.get(
                          new Integer(aPath[heightOfSearch-1]))).elements();
  while (W.hasMoreElements()) {
    justARouter = ((Integer)W.nextElement()).intValue();
    if (causeNoCycle(aPath, heightOfSearch, justARouter)) {

      // assign this router as the source in this path
      aPath[heightOfSearch] = justARouter;

      // record the new path id, etc.
      path_id = PIB.getNewPathId(justARouter, aPath[DESTINATIONNODE]);

      // run through the SLP's and record their sequence
      for (int index = heightOfSearch; index >DESTINATIONNODE; index--){

        // determine link_id of this SLP
        link_id = PIB.getLinkBetween(aPath[index],
                                     aPath[index-INCREMENTATIONOFSEARCH]);

        // assign the SLP its sequence number
        sequence_number = heightOfSearch - index;
        PIB.assignSLPSequence(service_level, aPath[index],
          link_id, path_id, sequence_number);
      }
      if (heightOfSearch < Hmax) {
        processPath(parent, aPath, heightOfSearch+INCREMENTATIONOFSEARCH,
                                                          service_level);
      }
    }
  }
  if (showComments){
    gui.sendText("Server: processPath: paths at depth of "+heightOfSearch
      +" from node "+aPath[DESTINATIONNODE]+" is completed.");
  }
}

/**
 * Checks to ensure that the addition of a specified new node to a specified
 * path does not result in a cycle being created. This check is completed by
 * the new node is already a member of the list of nodes in the path already.
 * @param aPath[] An array contain a path from a source node,
 * aPath[heightOfSearch], to a destination node, aPath[0].
```

```
 * @param heightOfSearch The number of nodes in the path so far.
 * @param justARouter The proposed next node in for a new path.
 * @returns noCycles True if no cycles are created by the addition of
 * justARouter.
 */
public boolean causeNoCycle(int aPath[], int heightOfSearch,
                            int justARouter){
  boolean noCycles = true;
  for (int index = INITIALZERO; index < heightOfSearch; index++){
    if (justARouter == aPath[index]){
      if (showComments){
        gui.sendText("Server: causeNoCycle: adding "+justARouter
          +" to get to "+aPath[DESTINATIONNODE]+" via "
          +aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
          +" at a height of "+heightOfSearch+" caused cycle!");
      }
      return noCycles = false;
    }
  }
  if (showComments){
    gui.sendText("Server: causeNoCycle: adding "+justARouter
      +" as hop #"+heightOfSearch+" to get to "+aPath[DESTINATIONNODE]
      +" via "+aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
      +" does not cause cycle.");
  }
  return noCycles;
}

/**
 * Determines what the effective QoS on each path in the PIB is. For each
 * path, the service level pipes that compose it are retrieved. Then, for
 * each of these service level pipes, we total up the delay and loss rate.
 * The effective throughput remaining is determined by finding the minimum
 * difference between the observed throughput and the target throughput of
 * each service level pipe.
 */
public void determineEffectiveQoSForPaths(){
  long start, finish;
  Vector path_ids;
  Integer myPathId;
  Vector SLPs;
  SLP mySLP;
  int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
    throughput = INITIALZERO, targetThroughput = INITIALZERO,
    throughputRemaining = INITIALZERO,
    minThroughputRemaining = INITIALZERO;

  // capture the start time of processing a path data
  start = System.currentTimeMillis();

  // for each path
  path_ids = PIB.getAllPathIds();

  for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){
```

```
    // for each path
    myPathId = (Integer)path_ids.elementAt(index1);

    SLPs = PIB.getSLPsOfPath(myPathId.intValue());

    for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

      mySLP = (SLP)SLPs.elementAt(index2);

      // add delay to total delay
      totalDelay = totalDelay + mySLP.getDelay();

      // add loss rate to total loss rate
      totalLossRate = totalLossRate + mySLP.getLossRate();

      // find min throughput
      throughput = mySLP.getThroughput();

      targetThroughput = mySLP.getTargetThroughput();

      throughputRemaining = targetThroughput - throughput;
      if (throughputRemaining < minThroughputRemaining ||
                              minThroughputRemaining == INITIALZERO){
        minThroughputRemaining = throughputRemaining;
      }

    }


    PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,
                                                    minThroughputRemaining);
    totalDelay = INITIALZERO;
    totalLossRate = INITIALZERO;
    minThroughputRemaining = INITIALZERO;
  }

  // capture the path data processing finish time
  finish = System.currentTimeMillis();
  gui.sendText("Server: determineEffectiveQoSForPaths: Time required = "
    +(finish-start)+" milliseconds.");

}

/**
 * Determines the effective QoS for just those paths that pass over the
 * specified service level pipe. For each path, the service level pipes that
 * compose it are retrieved. Then, for each of these service level pipes, we
 * total up the delay and loss rate. The effective throughput remaining is
 * determined by finding the minimum difference between the observed
 * throughput and the target throughput of each service level pipe.
 * @param address The address of the interface containing this service level.
 * @param service_level The service level of this SLP.
 */
public void determineEffectiveQoSForPaths(IPv6Address address,
                                          int service_level){
```

```
long start, finish;
Vector path_ids;
Integer myPathId;
Vector SLPs;
SLP mySLP;
int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
  throughput = INITIALZERO, targetThroughput = INITIALZERO,
  throughputRemaining = INITIALZERO, minThroughputRemaining = INITIALZERO;

// capture the start time of processing a path data
start = System.currentTimeMillis();

// for each path
path_ids = PIB.getAllPathIdsThatTraverseSLP(address, service_level);

for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

  // for each link
  myPathId = (Integer)path_ids.elementAt(index1);

  SLPs = PIB.getSLPsOfPath(myPathId.intValue());

  for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

    mySLP = (SLP)SLPs.elementAt(index2);

    // add delay to total delay
    totalDelay = totalDelay + mySLP.getDelay();

    // add loss rate to total loss rate
    totalLossRate = totalLossRate + mySLP.getLossRate();

    // find min throughput
    throughput = mySLP.getThroughput();

    targetThroughput = mySLP.getTargetThroughput();

    throughputRemaining = targetThroughput - throughput;
    if (throughputRemaining < minThroughputRemaining ||
                                    minThroughputRemaining == INITIALZERO){
      minThroughputRemaining = throughputRemaining;
    }

  }


  PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,
                                                  minThroughputRemaining);
  totalDelay = INITIALZERO;
  totalLossRate = INITIALZERO;
  minThroughputRemaining = INITIALZERO;
}

// capture the path data processing finish time
finish = System.currentTimeMillis();
```

```
    gui.sendText("Server: determineEffectiveQoSForPaths: Time required ="
      +(finish-start)+" milliseconds.");
  }
  /**
   * Returns the String representation of this Server.
   * @return The String representation of this Server.
   */
  public String toString(){
    return "Server";
  }  //

  //methods below are added by akkoc
 /**
   * Method for receiving required values from demosation for server settings
   * Also this method is used for server to place an entry for itself
   * in the servertable
   * @return void.
   */
  public synchronized void processConfiguration (Configuration con){
    serverType = con.getServerType();
    flowId = con.getFlowId();
    metricType = con.getmetricType();
    cycleTime = con.getCycleTime();
    globalTime = con.getGlobalTime();
    AutoConfigurationExecutive ace=controlExec.getAutoConfigurationExecutive();
    ace.createNewServerInformation(flowId,controlExec.getRouterId());

    initHeartbeat();//added by Efraim KATI (February 2000)

 } // end processConfigurtaion

  /**
   * Creates thread for dcm sending from the server.
   * @return void.
   */
  public void autoConfig() {
    Thread configThread = new Thread(this,"AutoConfig");
    configThread.start();
  }//end of autoconfig

  /**
   * Triggers DCM sending. and provides continues resreshment of SAAM region
   * with DCM messages.
   * @return void.
   */

  public void run(){
       gui.sendText("\n Server  will send first DCM after 40 secs");
       try{
     gui.sendText("thread is sleeping now ");
     Thread.sleep(60000);
     gui.sendText("thread woke up after 40  secs so start sending ");
   }catch(InterruptedException ie){}
```

```
  while(true) {

    try{

     Vector tableEntries = controlExec.getEmulationTable().getEmTable();
     Enumeration es = tableEntries.elements();
     while( es.hasMoreElements()){
       EmulationTableEntry ent = (EmulationTableEntry) es.nextElement();
       //destination adress determined from emulationtable entry
       IPv6Address des = new IPv6Address(ent.getNextHopIPv6().getAddress());
       //gui.sendText(" Destination of DCM is "+des.toString());
       byte[] nextHopBytes = des.getAddress();
       Vector interfaces = new Vector();
       interfaces = this.controlExec.getInterfaces();

       IPv6Address sInt;
       for(int i=0;i<interfaces.size();i++){
         Interface thisInterface = (Interface)interfaces.get(i);
         //cycle through all interfaces checking network address
         //against nextHop.
         int match = 0;
         byte[]
outboundInterfaceBytes=thisInterface.getID().getIPv6().getAddress();
         int bytesToCheck = 5;

         for(int index=0;index<bytesToCheck;index++){

if((nextHopBytes[index]&0xFF)==(outboundInterfaceBytes[index]&0xFF)){
             match++;
           }//if
         }//inner for

         if(match== bytesToCheck){
           sInt = new
IPv6Address(thisInterface.getID().getIPv6().getAddress());
           sendDown(sInt,des);
         }//if
       }//outer for

     }// end while
   }catch(UnknownHostException e){
     gui.sendText(e.getMessage()+"inside catch of DCM start up using em
table");
   }//try-catch

   try{
     Thread.sleep(this.cycleTime); //from demostation
   }catch(InterruptedException ie){
     gui.sendText("thread sleep problem");
   }

  }//end of while providing continues DCM sending

 }// end run()
```

```
/**
 * Retruns flowid of server.
 * @return ind serverflow id.
 */
public int getServerFlowId(){
  return flowId;
}

/**
 * Returns type of server(0-> for Primary, 1-> for Backup )
 * @return byte value.
 */
public byte  getServerType(){
   return serverType;
}

/**
 * Method to send the DCM message using controlExecutive sendDCM method
 * @return void.
 */
public void sendDown(IPv6Address srcInt,IPv6Address des) {

  DCM myDCM = new DCM(flowId,ServerId,metricType,srcInt,CTS,globalTime,
                                          getSeguenceNumberForDcmSending());
  gui.sendText("DCM with SQ is sent "+this.getSeguenceNumberForDcmSending());
  setSeguenceNumberForDcmSending();
  short sourcePort = ControlExecutive.SAAM_CONTROL_PORT;
  short destPort = ControlExecutive.SAAM_CONTROL_PORT;

  try{
     controlExec.sendDCM(this, myDCM, getServerFlowId(),
                         sourcePort,des, destPort);
     //gui.sendText("DCM has been sent");
  }catch(Exception fe){
     System.err.println(fe.toString());
  }

}//end sendDown()

/**
 * Method for setting proper value to put in DCM message for sequence
 * number field
 * @return void.
 */
private void setSeguenceNumberForDcmSending(){
  sequenceNumber++;
  if(sequenceNumber == 65535) sequenceNumber = 0;
}

 /**
 * Method for returning current sequence number value
 * @return int value.
 */
```

```
  private int getSeguenceNumberForDcmSending(){
    return  sequenceNumber;
  }


//*************************************************************************
//
// The following methods are added by Efraim KATI (February 2000)
//
//*************************************************************************

  /**
   * Receives the heartbeat query message coming from the backup server. After
   * that prepares the heartbeat response message with the same sequence number
   * and sends it to the backup server
   * @param hbq heartbeat query message
   * @return void
   */
   public void processHeartbeatQuery(HeartbeatQuery hbq){

      try{

         gui.sendText("\" Heartbeat Query \" is received." + "  Seq.Num : " +
                     hbq.getSequenceNumber()+
                     "   at : " + System.currentTimeMillis()+"" );
         hbr.setLastUsedFlowID(lastUsedFlow_ID);
         hbr.setSequenceNumber(hbq.getSequenceNumber());

         if(testCase2 &&
            (hbq.getSequenceNumber() == 3 || hbq.getSequenceNumber() == 4)){

            gui.sendText("\nTEST CASE #2 " +
                     "\" Heartbeat Response  \" did not send on purpose"+
                     "\nSeq.Num : " +
                     hbr.getSequenceNumber()+"" );

         }
         else if (testCase3 &&
                  (hbq.getSequenceNumber() == 11 ||
                   hbq.getSequenceNumber() == 12) ){
            gui.sendText("\nTEST CASE #3 " +
                     "\" Heartbeat Response  \" did not send on purpose"+
                     "\nSeq.Num : " +
                     hbr.getSequenceNumber()+"" );

         }
         else if (testCase3 &&
                  (hbq.getSequenceNumber() == 13 )){

            gui.sendText("\nTEST CASE #3 " +
                     "\" Heartbeat Response  \" sent with sequnce number "+
                     "\"3\" on purpose");
            hbr.setSequenceNumber((short)3);
            controlExec.send(this,
```

```
                        hbr,
                        4,
                        (short)PSUEDORANDOMSOURCEPORT,
                        IPv6Address.getByName(backupServerIPv6Str),
                        (short)8000);

        }
        else if (testCase4 &&
                 (hbq.getSequenceNumber() > 4) &&
                 (hbq.getSequenceNumber() < 9)){

           gui.sendText("\nTEST CASE #4 " +
                    "\" Heartbeat Response  \" did not send on purpose. "+
                    " SeqNum  :" +hbq.getSequenceNumber()+"" );

        }
        //normal case
        else{

           controlExec.send(this,
                            hbr,
                            4,
                            (short)PSUEDORANDOMSOURCEPORT,
                            IPv6Address.getByName(backupServerIPv6Str),
                            (short)8000);

           gui.sendText("*********************************************"+
                        "******************" +
                        "\" Heartbeat Response  \" is sent"+ "  Seq.Num : " +
                        hbr.getSequenceNumber()+
                        " at : " + System.currentTimeMillis()+"" );

        }//end if

     }//end try
     catch(FlowException fe){

       gui.sendText(fe.getMessage());

     }//end catch
     catch(UnknownHostException uhe){

       gui.sendText(uhe.getMessage());

     }//end try/catch

  }//end processHeartbeatQuery()


 /**
  * Receives the heartbeat response message coming from the primary server.
  * @param hbr heartbeat response message
  * @return void
  */
  public void processHeartbeatResponse(HeartbeatResponse hbr){
```

```
short seqNum = hbr.getSequenceNumber();

if(!isMainDown){

   gui.sendText(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"+
                "\" Heartbeat Response  \"  is received with"+
                " SeqNum. : " + seqNum +"  at :  "  +
                System.currentTimeMillis()+"");

   if(seqNum == lastSequenceNumber){

      lastResponseTime = System.currentTimeMillis();

      if(!hbController.getStatus()){

         hbController.restartQuerySendTimer();
         hbController.setStatus(true);
         //clear the vector because everything is normal again
         clearRecentMissedSequences();
         gui.sendText("vector is cleared:");
         printRecentMisses();

      }//end if

      hbController.stopResponseControlTimer();

   }
   else if (recentMissedSequences.contains(new Short(seqNum))){

      lastResponseTime = System.currentTimeMillis();
      gui.sendText("\n!!!!!! Received HeartbeatResponce Sequence Number"+
                   " did not match ");
      gui.sendText("Received sequence was   :   " +
                   hbr.getSequenceNumber()+ "");
      gui.sendText("Expected sequence was   :   " +
                   lastSequenceNumber+ "");
      gui.sendText("Sequence Number did not match but it exists in "+
                   "the recent misses. So it is accepted.");
      if(!hbController.getStatus()){

         hbController.restartQuerySendTimer();
         hbController.setStatus(true);
         //clear the vector because everything is normal again
         clearRecentMissedSequences();
         gui.sendText("vector is cleared:");
         printRecentMisses();

      }//end if

      hbController.stopResponseControlTimer();
   }
   else {

      gui.sendText("\n!!!!! Received HeartbeatResponce Sequence Number"+
```

```
                    " did not match ");
          gui.sendText("Received sequence was   :   " +
                    hbr.getSequenceNumber()+ "");
          gui.sendText("Expected sequence was   :   " +
                    lastSequenceNumber+ "");
          gui.sendText("Sequence Number did not match and it not in the "+
                    "recent misses. So it is ignored.");
          printRecentMisses();

       }//end if

    }//end if

 }//end processHeartbeatResponse()

/**
 * Returns the value of the lastUsedFlowID data member
 * @param none
 * @return lastUsedFlow_ID
 */
 public int getLastUsedFlowID(){

    return lastUsedFlow_ID;

 }//end getLastUsedFlowID()

/**
 * Sends a heartbeat query message to the primary server
 * @param none
 * @return void
 */
 public void sendHeartbeatQuery() {

    HeartbeatQuery hbq = new HeartbeatQuery();
    lastSequenceNumber = hbq.getSequenceNumber();

    try{
       controlExec.send(this,
                    hbq,
                    2,
                    (short)PSUEDORANDOMSOURCEPORT,
                    IPv6Address.getByName(primaryServerIPv6Str),
                    (short)8000);
       currentTime = System.currentTimeMillis();
       long timeDiff =  currentTime -  lastQueryTime;
       gui.sendText("\" Heartbeat Query \" is sent with "+
                 "SeqNum. : " + lastSequenceNumber + "  at : " +
                 currentTime+"  after  :"  + timeDiff + " milliseconds");
       lastQueryTime = currentTime;

    }//end try
    catch(FlowException fe){

       gui.sendText(fe.getMessage());
```

```
      }//end catch
      catch(UnknownHostException uhe){

        gui.sendText(uhe.getMessage());

      }//end try/catch

      hbq = null;
      System.gc();
      hbController.restartResponseControlTimer();

   }//end sendHeartbeatQuery()

  /**
   * Sets the value of the isMainDown boolean
   * @param status
   * @return void
   */
   public void setIsMainDown(boolean status){

      isMainDown = status;

      if ( !isMainDown){

         bf.setVisible(false);

      }//end if

   }//end setIsMainDown()

  /**
   * Adds the sequence number of the last missed heartbeat response message to
   * recentMissedSequences Vector
   * @param none
   * @return void
   */
   public void addRecentMissedSequences(){

      recentMissedSequences.addElement(new Short(lastSequenceNumber));
      gui.sendText("Sequence Nubmer : " + lastSequenceNumber  +
                   " is added to the \"RecentMissedSequences\"vector");
      printRecentMisses();

   }//end addRecentMissedSequences()

  /**
   * This method deletes all of the elements in the recentMissedSequences
Vector
   * @param none
   * @return void
   */
   public void clearRecentMissedSequences(){

      recentMissedSequences.removeAllElements();
```

```
    }//end clearRecentMissedSequences()

    /**
     * Displays the content of the recentMissedSequences Vector to the screen
     * @param none
     * @return void
     */
    public void printRecentMisses(){

        Enumeration e = recentMissedSequences.elements();
        gui.sendText("\"RecentMissedSequences\" Vector elements:  ");
        Short a;

        while (e.hasMoreElements()){

            a = (Short)e.nextElement();
            gui.sendText(""+ a.shortValue() +"");

        }//end while

    }//end printRecentMisses()

    /**
     * Returns the value of the lastResponseTime data member
     * @param none
     * @return lastResponseTime
     */
    public long getLastResponseTime(){

        return lastResponseTime;

    }//end getLastResponseTime()

    /**
     * Returns the last DCM reception time
     * @param none
     * @return lastT
     */
    public long getLastDCMTime(){

        long lastT=controlExec.getAutoConfigurationExecutive().getLastDCMTime();
        return lastT;

    }//end getLastDCMTime()

    /**
     * Display the passed string on the screen
     * @param str string to be displayed
     * @return void
     */
    public void display(String str){

        gui.sendText(str);

    }//end display()
```

```
/**
 * Initializes the banner frame according to the server type. If this is the
 * backup server, then it starts the heartbeat query sending process
 * @param none
 * @return void
 */
private void initHeartbeat(){

   if(serverType == (byte)0){

      isMain = true;
      bf.setFrameText("THIS IS THE PRIMARY SERVER");
      bf.setVisible(true);
      gui.setTextField("The primary Server is active rigth now");

   }
   else{

      isMain = false;
      bf.setBackgroundColor(Color.cyan);
      bf.setFrameText("THIS IS THE BACKUP SERVER");
      bf.setVisible(true);
      gui.setTextField("The Backup Server is silent rigth now");
      hbController = new HeartbeatController(this, tmax, tmin, bf);
      //waits for to start sending HeartbeatQuery Messages

      Timer startHeartbeatQueryTimer = new Timer(firstQueryTime,
            (new java.awt.event.ActionListener(){
               public void actionPerformed(ActionEvent e){
                  hbController.startSending();
                  gui.sendText("querySendTimer  is started at :" +
                  System.currentTimeMillis() + "");
                  gui.sendText("First Heartbeat Query Message will "+
                              "be sent after  " +
                              firstQueryTime  + " milliseconds");
               }
            }));

      startHeartbeatQueryTimer.setRepeats(false);
      startHeartbeatQueryTimer.start();

   }//end if

   //****************************
   // server table did not return correct value. Therefore server
IPv^Address
   // are hardcoded and this section is commented
   //****************************
   /*
   ServerTable serverTable = controlExec.getServerTable();
   Vector serverTableVector = serverTable.getTable();
   ServerTableEntry serverTableEntry = null;
   Enumeration e = serverTableVector.elements();
   int flowID = 0;
```

```
      IPv6Address addr=null;
      while(e.hasMoreElements()){

         serverTableEntry = (ServerTableEntry)e.nextElement();
         flowID = serverTableEntry.getFlowId();
         addr = serverTableEntry.getServerAddress();

         if(flowID==1){

            primaryServerIPv6Str = addr.toString();

         }//end if

         if(flowID==3){

            backupServerIPv6Str = addr.toString();

         }//end if

      }//end while

      gui.sendText("Primary Server IPv6Address:  " +
                   primaryServerIPv6Str );

      gui.sendText("Backup Server IPv6Address:  " +
                   backupServerIPv6Str );

    */
   }//end initHeartbeat()

}//end Server class

//end file Server.java
```

```
//---------------------------------------------------------------------------
// Filename      : ServerAgent.java
// Date          : December 8, 1999
// Author        : John YARGER (August 1999)
// Modified by   : Hasan AKKOC (February 2000)
// Modified by   : Efraim KATI (February 2000)
// Project       : SAAM
//---------------------------------------------------------------------------


package saam.residentagent.server;

import saam.control.*;
import saam.residentagent.*;
import saam.server.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.event.SaamListener;

public class ServerAgent implements ResidentAgent,
                                                MessageProcessor,
                                                SaamListener{

  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private Server myServer;

  private String[] messageTypes =
    {"saam.message.Hello",
     "saam.message.FlowRequest",
     "saam.message.LinkStateAdvertisement",
     "saam.message.Configuration",
     "saam.message.HeartbeatQuery",
     "saam.message.HeartbeatResponse"};


  public void install(ControlExecutive controlExec){
    gui=new SAAMRouterGui("ServerAgent");
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
    myServer = new Server("classObject", controlExec);
    gui.sendText("\nCalling My Server method: autoconfig()");
    myServer.autoConfig();
  }

  public void processMessage(Message message){
    try{
      if(message instanceof Hello){
        gui.sendText("Received Message: "+((Hello)message));
        gui.sendText("Calling Server method: processHello()");
        myServer.processHello((Hello)message);
      }else if(message instanceof FlowRequest){
        FlowRequest request = (FlowRequest)message;
        gui.sendText("Received Message: "+ request);
```

```
      gui.sendText(
        "Calling Server method: processFlowRequest()");
      myServer.processFlowRequest((FlowRequest)message);

    }else if(message instanceof LinkStateAdvertisement){
      gui.sendText("Received Message: "+
        ((LinkStateAdvertisement)message));
      gui.sendText("Calling Server method: processLSA()");
      myServer.processLSA(
        (LinkStateAdvertisement)message);
    }
    //below added by akkoc
    else if(message instanceof Configuration){
      gui.sendText("Received Message: "+
        ((Configuration)message));
      gui.sendText("Calling Server method: processConfiguration()");
      myServer.processConfiguration((Configuration)message);
    }
    //the foolowing two "else if" cases are added by Efraim KATI
    else if(message.getType() == (byte)2){
      gui.sendText("Received Message: "+ ((HeartbeatQuery)message));
      gui.sendText("Calling Server method: processHeartbeatQuery()");
      myServer.processHeartbeatQuery((HeartbeatQuery)message);
    }

    else if(message.getType() == (byte)3){
      gui.sendText("Received Message: "+ ((HeartbeatResponse)message));
      gui.sendText("Calling Server method: processHeartbeatResponse()");
      myServer.processHeartbeatResponse((HeartbeatResponse)message);
    }

  }catch(Exception e){message = null;}
}

public String[] getMessageTypes(){
  gui.sendText("Server queried my message types");
//  gui.sendText("Sending: "+messageTypes[0]);
  return messageTypes;
}
public String toString() {
  String it = "ServerAgent listening for: ";
  for(int i=0;i<messageTypes.length;i++){
    it += "\n" + messageTypes[i];
  }
  return it;
}//toString()

//the following methods are stubbed out as they are not used.
public void uninstall(){
}
public Message query(Message message){
  return message;
}
public void transferState(ResidentAgent replacement){
}
```

```
    public void receiveState(Message message){
    }
    public void receiveFlowResponse(FlowResponse flowResponse){
    }

}//end ServerAgent class

//end file ServerAgent.java
```

```
//-----------------------------------------------------------------------------
// Filename     : PacketFactory.java
// Date         : August 1st, 1999
// Author       : Dean VRABLE (August 1999)
// Modified by  : Hasan AKKOC (February 2000)
// Modified by  : Efraim KATI (February 2000)
// Project      : SAAM
//-----------------------------------------------------------------------------

package saam.control;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;
import java.util.TooManyListenersException;
import java.util.StringTokenizer;
import java.lang.reflect.Constructor;

import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.residentagent.*;

/**
* A PacketFactory can be used to build SaamPackets for sending or
* to receive SaamPackets and extract their atomic elements.  These
* atomic elements are currently one of two types:  A subclass of
* saam.residentagent.ResidentAgent or a subclass of
* saam.message.Message.<p>
* A sender would instantiate a PacketFactory to build
* Saam Packets.  The PacketFactory's append methods receive
* Message Objects, ResidentAgent Objects, or a String that represents
* the class name of a ResidentAgent as parameters and then dynamically
* construct the appropriate header based on the number of elements
* received and the current time.  The getBytes method is used to
* retrieve the byte array that represents the SAAMPacket that has been
* constructed by this PacketFactory.<p>
* The ControlExecutive uses the PacketFactory to receive and parse
* SaamPackets.
*/
public class PacketFactory extends Thread
  implements SaamTalker, SaamListener{

  private final boolean guiActive = true;
  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private boolean started = false;
  private boolean firstEvent = true;
  private boolean bytesRetrieved;
  private byte[] packet,DCMpacket,PNpacket,UCMpacket;
```

```
private byte numberOfMessages;
private Loader loader;
private Class message;
private SaamEvent currentEvent;
private Thread owner;
//private static int instanceNumber;
private Object theLock = new Object();

// xie
private FIFOQueue inputQueue = new FIFOQueue(1000);

/**
 * Use the no-args constructor to begin constructing packets
 * on the sending side.
 */
//no-args constructor doesn't come for free when we have
//another constructor
public PacketFactory(){
  //instanceNumber++;
  //gui = new SAAMRouterGui(toString() + "("+ instanceNumber+")");
  gui = new SAAMRouterGui("Output "+ toString());
  gui.setTextField("I construct outbound packets");
}

/**
 * This constructor is not available to Objects outside the
 * saam.control package.  The ControlExecutive uses this constructor
 * to receive and parse SAAMPackets.  The PacketFactory passes the
 * atomic elements (either ResidentAgents or Messages) up to the
 * ControlExecutive for further processing.
 * @param controlExec The ControlExecutive that is to receive
 * updates from this PacketFactory.
 */
PacketFactory(ControlExecutive controlExec){
  //this();
  gui = new SAAMRouterGui("Input " + toString());
  gui.setTextField("I Listen for inbound packets");
  this.controlExec=controlExec;
  loader = new Loader();

  //*****************************
  //**Listen to desired Channels**
  //*****************************
  int channel_ID =
    ProtocolStackEvent.PACKETFACTORY_CHANNEL;
  try{
    controlExec.addListenerToChannel(this, channel_ID);
    gui.sendText("Listening to channel: "+channel_ID);
  }catch(ChannelException ce){
    gui.sendText(ce.toString());
  }//try-catch

  //***************************************
  //**Register to talk on desired Channels**
  //***************************************
```

```
    channel_ID = ControlExecutive.SAAM_CONTROL_PORT;
    try{
       controlExec.addTalkerToChannel(this,
         channel_ID);
       gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
       gui.sendText(ce.toString());
    }

    start();
  }

  /**
   * When instantiated to receive packets, the PacketFactory
   * Thread waits until a SAAMPacket arrives, then it calls
   * the processPacket method.
   */
  public void run(){
     while(true){
       gui.sendText("\n Inside PacketFactory run()");
       synchronized (theLock){
         if (inputQueue.isEmpty()){
           started = false;
           try{
             gui.sendText("Waiting...");
             theLock.wait();
             gui.sendText("Continuing");
           }
           catch(InterruptedException e){
             gui.sendText("Interrupted exception catched");
           }
         }// end if

         packet = (byte[]) inputQueue.dequeue();

       }// end synchronization

       processPacket();

     }//while(true)
  }

  /**
   * This method is called by the Channels this Object has registered to
   * monitor when a talker sends events on those Channels.
   * @param se The SaamEvent to be communicated.

  public synchronized void receiveEvent(SaamEvent se){
 */
  public  void receiveEvent(SaamEvent se){

    gui.sendText("\n---- Got a packet");
    currentEvent = se;
    ProtocolStackEvent psec = (ProtocolStackEvent)currentEvent;
```

```
  byte[] newcomer = psec.getPacket();
  gui.sendText("\n New packet has length = " + newcomer.length);

  synchronized(theLock){
    inputQueue.enqueue((Object) newcomer);
    if (!started){
      started = true;
      gui.sendText("\n Waking up the processPacket thread");
      theLock.notify();
    }// end if
  }
}

/**
 * This method is used to extract the individual Class
 * Objects that are represented in the packet.  These Class
 * Objects are either of type 0 (ResidentAgent) or 1 (Message).<p>
 * If a ResidentAgent is received, a Class Object is created
 * that represents the agent.  That Class Object is then sent to
 * the ControlExecutive for screening and agent instantiation.<p>
 * If a Message is received, that Message is instantiated and sent
 * to the ControlExecutive for further processing.
 */

private void processPacket() {
  int channel = currentEvent.getChannel_ID();
  String eventSource = (String)currentEvent.getSource();

  //see saam.util for PrimitiveConversions and Array classes
  long timeStamp = PrimitiveConversions.getLong(
      Array.getSubArray(packet,0,8));
  numberOfMessages=packet[8];
  gui.sendText("packet arrived: " +
    "\n  source:       " + eventSource +
    "\n  channel:      " + channel +
    "\n  size:         " + packet.length +
    "\n  # of Messages: " + numberOfMessages +
    "\n  timeStamp:    " + timeStamp);

  //now we trim the packet by removing the header.
  packet = Array.getSubArray(packet,9,packet.length);

  //used to track the current position in the array.
  int index = 0;

          for(int i=1;i<=numberOfMessages;i++){
    gui.sendText("\nProcessing Element["+i+"]:");
  //      int index = 0;
            byte type = packet[index++];

    switch(type){
      case 0:
      case 1:
                //extract and process each atomic element of the packet
                //separately.  Here we assume the packet is a properly
```

```
        //formatted SAAMPacket when it arrives, and that the
        //length is less than the max allowed.

        gui.sendText("  type:   "+type);
        //retrieve the number of bytes the class name occupies
        byte nameLength = packet[index++];

        //extract the name of the class file as a byte array
        byte[] elementNameArray = Array.getSubArray(
          packet,index, index+nameLength);
        index+=nameLength;

        //convert the name back into a String
        String elementName = new String(elementNameArray);
        gui.sendText("  Name: "+elementName);

        //retrieve the length of the Object
        short length = PrimitiveConversions.getShort(
          Array.getSubArray(packet,index,index+2));
        gui.sendText("  Length:       "+length);
        index+=2;

        //retrieve the bytecode of the Object
        byte[] bytes = Array.getSubArray(
          packet,index,index+length);
        index+=length;

                    if (type == 0){
gui.sendText("This is a ResidentAgent");
      //Assume this class is of type ResidentAgent
      try{
        //Attempt to define the class using the current
        //class loader.
        loader.defClass(elementName, bytes);
      }catch(LinkageError le){
        //If the loader already has a definition for the class
        //a LinkageError will be thrown.  If this happens, we
        //need to instantiate a new class loader and use it to
        //define the class.  A nice little trick we learned from
        //page 55 of Jason Hunter's "Java Servlet Programming" book.
        gui.sendText(le.toString());
        gui.sendText("Class was previously loaded...");
        gui.sendText("Replacing old ClassLoader...");
        Loader newLoader = new Loader();
        newLoader.defClass(elementName, bytes);
      }
      try{
        //message is of type Class.
        message = Class.forName(elementName, true, loader);
      }catch(ClassNotFoundException cnfe){
        gui.sendText(cnfe.toString());
      }
      gui.sendText(message.toString());
      ResidentAgentEvent rae = new ResidentAgentEvent(
        eventSource,
```

```
            this,
            ControlExecutive.SAAM_CONTROL_PORT,
            message);
          try{
            gui.sendText("Forwarding on channel "+
              ControlExecutive.SAAM_CONTROL_PORT);
            controlExec.talk(rae);
          }catch(ChannelException tde){
            gui.sendText(tde.toString());
          }
                        }else {
          gui.sendText("This is a Message");
          //Assume this class is of type Message.
          try{
            //message is of type Class.
        message = Class.forName(elementName);
          }catch(ClassNotFoundException cnfe){
        System.out.println("error is here  ");
            {gui.sendText("Bytecode for: "+elementName+
              " not found.");
            }
          }

          try{
            //Call the constructor from within this Class that
            //takes a byte array as its only argument
            Constructor cons = message.getConstructor(
              new Class[] {byte[].class});

            //Create the instance of this Message
            Message instance = (Message)cons.newInstance(
                new Object[] {bytes});
            gui.sendText(instance.toString());
            MessageEvent me = new MessageEvent(eventSource, this,
              ControlExecutive.SAAM_CONTROL_PORT,  instance);
            //send this MessageEvent on the Control port.
            try{
              gui.sendText("Forwarding on channel "+
                ControlExecutive.SAAM_CONTROL_PORT);
              controlExec.talk(me);
            }catch(ChannelException tde){
          gui.sendText("problem occured here ");
              gui.sendText(tde.toString());
            }
          }catch(Exception e){
            //need to notify sender that we have no classfile
            //with this name
            gui.sendText(e.toString());
          }//try-catch
                        }
    break;

  //THIS CASE PROCESSES THE HEARBEATQUERY MESSAGE
  case 2:
```

```
        //retrieve the bytecode of the Object
        bytes = Array.getSubArray(packet,1,packet.length);
        gui.sendText("This is a  HeartbeatQuery message");

        //Create the instance of this Message
        HeartbeatQuery hbq = new HeartbeatQuery(bytes);
        gui.sendText(hbq.toString());

        MessageEvent hbqMe = new MessageEvent(eventSource,
                                              this,
                                              ControlExecutive.SAAM_CONTROL_PORT,
                                              hbq);
        //send this MessageEvent on the Control port.
        try{
          gui.sendText("Forwarding on channel "+
          ControlExecutive.SAAM_CONTROL_PORT);
          controlExec.talk(hbqMe);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }

        break;

      //THIS CASE PROCESSES THE HEARBEATRESPONSE MESSAGE
      case 3:

        //retrieve the bytecode of the Object
        bytes = Array.getSubArray(packet,1,packet.length);
        gui.sendText("This is a  HeartbeatResponse message");

        //Create the instance of this Message
        HeartbeatResponse hbr = new HeartbeatResponse(bytes);
        gui.sendText(hbr.toString());

        MessageEvent hbrMe = new MessageEvent(eventSource,
                                              this,
                                              ControlExecutive.SAAM_CONTROL_PORT,
                                              hbr);
        //send this MessageEvent on the Control port.
        try{
          gui.sendText("Forwarding on channel "+
          ControlExecutive.SAAM_CONTROL_PORT);
          controlExec.talk(hbrMe);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }

        break;

      case 4:
        gui.sendText("This is a DCM  Message");

          try{
            DCM  dcm = new DCM(packet);
            gui.sendText(dcm.toString());
```

```
    MessageEvent me = new MessageEvent( eventSource, this,
                                ControlExecutive.SAAM_CONTROL_PORT,dcm);
    //send this MessageEvent on the Control port.
    try{
      gui.sendText("Forwarding on channel "+
                ControlExecutive.SAAM_CONTROL_PORT);
                controlExec.talk(me);

    }catch(ChannelException tde){
      gui.sendText(tde.toString());
      tde.printStackTrace();
    }
  }catch(Exception e){
    gui.sendText(e.toString());
    e.printStackTrace();
}//try-catch
break;

case 5:
  gui.sendText("This is a UCM  Message");
  try{
    UCM ucm  =  new UCM(packet);
    gui.sendText("\n"+ucm.toString());
    MessageEvent me = new MessageEvent( eventSource, this,
          ControlExecutive.SAAM_CONTROL_PORT,ucm);
    //send this MessageEvent on the Control port.
    try{
      gui.sendText("Forwarding on channel "+
         ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(me);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
  }catch(Exception e){
    gui.sendText(e.toString());
}//try-catch
break;

case 6:
  gui.sendText("This is a ParentNotification Message");
  //Assume this class is of type Message.
  try{
    ParentNotification pn = new ParentNotification(packet);
    MessageEvent me = new MessageEvent( eventSource,
                                this,
                                ControlExecutive.SAAM_CONTROL_PORT,
                                pn);
    //send this MessageEvent on the Control port.
    try{
      gui.sendText("Forwarding on channel "+
      ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(me);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
```

```
          }catch(Exception e){
            gui.sendText(e.toString());
        }//try-catch
        break;

        default:
          gui.sendText("Packet type unrecognized: "+type);
          //packet type is unrecognized.  Here we could
          //extract a channel_ID that could be embedded
          //in the packet, and then send the unrecognized
          //element on that channel.
    }//end switch
  }//for
}//processPacket()

/**
 * This method can be used to append a Message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param me The Message to be appended.
 */
public void append(Message me){

  if(bytesRetrieved){
    packet=null;
    numberOfMessages=0;
    bytesRetrieved = false;
  }
  byte type = me.getType();
  byte[] parameters = me.getBytes();
  //here we could check the length of the parameter array supplied
  //with the length returned from the length() method call.
  short paramLength = (short)parameters.length;


  //IN THIS SWITCH STATEMENT CASE 1 IS DESIGNED TO SUPPORT THE
  //OLD VERSION MESSAGE TYPE
  //
  //CASE 1 :  FOR OLD MESSAGE TYPE(TYPE=1)
  //CASE 2 :  FOR THE HEARTBEATQUERY TYPE(TYPE=2)
  //CASE 3 :  FOR THE HEARTBEATRESPONSE TYPE(TYPE=3)
  //
  //OTHER CASES WILL BE IMPLEMENTED FOR OTHER MESSAGE TYPES
  switch(type){

    case 1://FOR OLD VERSION MESSAGE TYPE

      String name = me.getClass().getName();
      byte nameLength = (byte)name.getBytes().length;


      //now append the Message to the packet byte array
      packet = Array.concat(packet,type);
      packet = Array.concat(packet,nameLength);
      packet = Array.concat(packet,name.getBytes());
```

```
      packet = Array.concat(packet,
        PrimitiveConversions.getBytes(paramLength));
      packet = Array.concat(packet,parameters);

      gui.sendText("Appended Message:" +
        "\n  Type:          " + type +
        "\n  name:          " + name +
        "\n  param length:  " + paramLength +
        "\n  # of elements: " + numberOfMessages +
        "\n  packet length: " + packet.length+"\n");

      break;


    case 2://FOR THE HEARTBEATQUERY MESSAGE

      packet = Array.concat(packet,type);
      packet = Array.concat(packet,parameters);
      gui.sendText("Appended Message:" +
        "\n  Type:          " + type +
        "\n  name:          " + "HeartbeatQuery" +
        "\n  param length:  " + parameters.length +
        "\n  packet length: " + packet.length+"\n");

      break;

    case 3://FOR THE HEARTBEATRESPONSE MESSAGE

      packet = Array.concat(packet,type);
      packet = Array.concat(packet,parameters);
      gui.sendText("Appended Message:" +
        "\n  Type:          " + type +
        "\n  name:          " + "HeartbeatResponse" +
        "\n  param length:  " + parameters.length +
        "\n  packet length: " + packet.length+"\n");
      gui.sendText(((HeartbeatResponse)(me)).toString());
      break;


    default:

      gui.sendText("Packet type unrecognized: "+type);

  }//end switch

  //increment the count of messages in this packet
  numberOfMessages++;

}  //end of append

 /**
 * This method can be used to append a DCM message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getDCMBytes method.
 * @param downWard The DCM message to be appended.
```

```
 */

public void appendDCM( DCM downWard){
DCMpacket = null;
  gui.sendText("Appending a dcm message before sending downward with lengh");
  DCMpacket = Array.concat(DCMpacket,downWard.getBytes());
  gui.sendText("Appending a dcm message before sending downward with lenght"+
               " " + DCMpacket.length);
}//end of appendDCm

/**
 * This method can be used to append a PN message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getPNBytes method.
 * @param downWard The PN message to be appended.
 */
public void appendPN( ParentNotification pn){
PNpacket =null;
  gui.sendText(" Appending a PN message before sending downward with lengh");
  PNpacket = Array.concat(PNpacket,pn.getBytes());
  gui.sendText("after appending PN is "+PNpacket.length);

}//end of appendDCm

/**
 * This method can be used to append a PN message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getPNBytes method.
 * @param downWard The PN message to be appended.
 */
public void appendUCM( UCM upWard){
  UCMpacket =null;
  gui.sendText(" Appending a UCM message before sending upward");
  UCMpacket = Array.concat(UCMpacket,upWard.getBytes());
}//end of appendUCM

/**
 * This method can be used to append a ResidentAgent to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param ra The ResidentAgent to be appended.
 */
public void append(ResidentAgent ra) throws IOException{
  String name = ra.getClass().getName();
  append(name);
}
/**
 * This method can be used to append a ResidentAgent by name to an
 * outgoing SAAMPacket.  To later retrieve the entire packet
 * (with header) as a byte array, call the getBytes method.
 * @param residentAgentClassName The String name of the ResidentAgent
 * classfile to be appended.
 */
public void append(String residentAgentClassName)
  throws IOException{
```

```
  if(bytesRetrieved){
    packet=null;
    numberOfMessages=0;
    bytesRetrieved = false;
  }
  byte type = 0;
  String name = residentAgentClassName;
  String fileName = "C:\\efraim"+File.separatorChar +
    residentAgentClassName.replace('.',File.separatorChar);
  fileName+=".class";
  gui.sendText("File name: "+fileName);
  FileInputStream fis = null;
  try{
    fis = new FileInputStream(fileName);
  }catch(IOException ioe){
    throw new IOException(
      "Problem reading ResidentAgent: "+fileName);
  }
  byte nameLength = (byte)name.getBytes().length;
  byte[] byteCode = new byte[fis.available()];
  short length = (short)fis.read(byteCode);

  packet = Array.concat(packet,type);
  packet = Array.concat(packet,nameLength);
  packet = Array.concat(packet,name.getBytes());
  packet = Array.concat(packet,
    PrimitiveConversions.getBytes(length));
  packet = Array.concat(packet,byteCode);
  numberOfMessages++;

  gui.sendText("Appended ResidentAgent:" +
    "\n  Type:              " + type +
    "\n  name:              " + name +
    "\n  byteCode length:   " + length +
    "\n  # of elements:     " + numberOfMessages +
    "\n  packet length:     " + packet.length+"\n");

}

/**
 * Appends a header to the byte array.  The header conforms
 * to the structure of a SAAMHeader.
 */
private void appendHeader(){
  byte[] timeStamp = PrimitiveConversions.getBytes(
    System.currentTimeMillis());
    packet = Array.concat(numberOfMessages,packet);
    packet = Array.concat(timeStamp,packet);
    gui.sendText("Appended header:"+
      "\n  timeStamp:     "+PrimitiveConversions.getLong(
        Array.getSubArray(packet,0,8))+
      "\n  # of updates: "+packet[8] +
      "\n  packet length: "+packet.length+"\n");

}
```

```
/**
 * Returns a byte array that conforms to the structure of
 * a SAAMPacket.
 * @return A byte array that conforms to the structure of
 * a SAAMPacket.
 */
public byte[] getBytes(){
  appendHeader();
  bytesRetrieved = true;
  return packet;
}

/**
 * Returns a byte array that conforms to the structure of a DCMPacket.
 * @return A byte array that conforms to the structure of DCMPacket.
 */

 public byte[] getDCMBytes(){
  return DCMpacket;
}

/**
 * Returns a byte array that conforms to the structure of a PNPacket.
 * @return A byte array that conforms to the structure of PNPacket.
 */
public byte[] getPNBytes(){
  return PNpacket;
}

/**
 * Returns a byte array that conforms to the structure of a UCMPacket.
 * @return A byte array that conforms to the structure of UCMPacket.
 */
public byte[] getUCMBytes(){
  return UCMpacket;
}

/**
 * Returns the current length of the packet.
 * @return The current length of the packet.
 */
public int length(){
  try{
    return packet.length;
  }catch(NullPointerException npe){
    return 0;
  }
}

/**
 * Returns a <code>String</code> representation of this object
 * @return The <code>String</code> representation of this object
 */
public String toString(){
```

```
    return "Packet Factory";
  }

}
```

LIST OF REFERENCES

1. Pankaj J., *Fault Tolerance in Distributed Systems,* pp.4-17, Prentice-Hall, 1994.

2. Gray, J., and Siewiorek, D. P., "High-Availability Computer Systems," *IEEE/Computer*, v. 24, no. 9, pp.39-48, September 1991.

3. Laprie, J. C., "Dependable Computing and Fault Tolerance: Concepts, and Terminology," *Proceedings of Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 2-11, June 1995.

4. Pradhan, D. J., *Fault-Tolerant Computer System Design,* pp.2-58, Prentice-Hall, 1996.

5. Microsoft Corporation, *Reliability and Fault Tolerance in Windows NT Server,* [http://www.microsoft.com/NTServer/fileprint/exec/overview/reliability.asp], September 1999.

6. "Deploying Microsoft Windows NT Server for High Availability", Microsoft Corporation, [http://microsoft.com/windows/news/notdated/highavailability.asp], October 1998.

7. Microsoft Corporation, *Windows NT 4.0 Resource Kit Documentation.*

8. Shnier. M., *Computer Dictionary*, p.526, Que Corporation, 1998.

9. Microsoft Corporation, *Microsoft Windows NT Network Administration,* pp.401, Microsoft Press, 1998.

10. Pfister, G., *In Search of Clusters*, pp.72, 385, Prentice-Hall, 1995.

11. Microsoft Corporation, *Clustering Architecture*, [http://www.microsoft.com/ntserver/ntserverenterprise/techdetails/prodarch/clustarchit.asp?RLD=36], April 1999.

12. Microsoft Corporation, *Microsoft Windows NT Clusters (White Paper)*, 1997.

13. Microsoft Corporation, *Clustering Overview,* [http://www.microsoft.com/ntserver/ntserverenterprise/exec/overview/Clustering.asp], July 1999.

14. Microsoft Corporation, *Cluster Strategy: High Availability and Scalability with Industry-Standard Hardware*, [http://www.microsoft.com/ntserver/ntserverenterprise/exec/prodstrat/cluster2.asp] , April 1999.

15. Microsoft Corporation, *Microsoft Windows NT Load Balancing Service (WBLS) Technical Overview*, [http://www.microsoft.com/ntserver/ntserverenterprise/techdetails/prodarch/Wlbs.asp], April 1999.

16. Patterson, D.A., Chen, P., Gibson, G., and Katz, R.H., "Introduction to redundant arrays of inexpensive disks (RAID)," *Thirty-fourth IEEE Computer Society International Conference: Intellectual Leverage*, pp. 112-117, February 1989.

17. Microsoft Corporation, *Windows NT Load Balancing Service (WLBS) Features Overview*, [http://www.microsoft.com/ntserver/ntserverenterprise/exec/feature/WLBS/WlbsFeat.asp], April 1999.

18. NSI Software Corporation, Theory of Operations Double-Take Version 3.0 for Windows NT and Solaris, [http://www.sunbelt-software.com/evals/dtake/Double-Take%20Theory%20of%20Operations.pdf], September 1999.

19. NSI Software Corporation, *High Availability for TCP/IP Networks*, [http://www. nsisw.com/pages/dt3napp.htm#wan], September 1999.

20. Legato Systems, *Octopus User's Guide Release 3.2*, [http://www.legato.com/support.index.html], 1999.

21. Legato Systems, *Co-StandbyServer for Windows NT (white paper)*, [http://kb.vinca.com/libraries/whitepapers/ntco_wp.pdf], 1999.

22. Legato Systems, *Co-StandbyServer for Windows NT User's Guide*, [http://software.vinca.com/software/ntco_man.pdf], 1998.

23. Marathon Technologies Corporation, Assured Availability Systems (white paper), [http://ginko.longwoodsw.com/svr/intrasrv.isv?mtc/mtcpublic.jfm], November 1998.

24. Computer Associates International, Inc., *ARCserve Replication 4.0 for Windows NT User Guide*, [http://www.cai.com/products/arcserve_replication/], 1998.

25. Computer Associates International, Inc., *High Availability Data Protection with ARCserve Replication For Windows NT (White Paper)*, [http://www.cai.com/products/arcserve_replication/replication_white_paper.htm], October 1999.

26. Gouda, M. G., and McGuire, T. M., "Accelerated Heart Protocols," *Proceedings of 18th International Conference on Distributed Computing Systems,* pp. 202-209, May 1998.

27. Barborak, M., Malek, M., and Dahbura, A., "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, v. 25, no. 2, pp. 171-220, June 1993.

28. Barry, N., "An end to downtime," [http://www.nwfusion.com/revievs/0705rev.html], July 1999.

29. Joseph, M., "Widespread High Availability Use Remains on Standby," [http://www.entmag.com/displayarticle.asp?searchresult=1&ID=1289871928PM] , December 1998.

30. Jonathan, C., "Easy-to-use solutions for uninterrupted data access," [http://www.winntmag.com/Magazine/Ereprint.cfm?ArticeID=3579], July 1998.

31. Carlos, B., "Mirroring Software to Prevent Disaster," [http://www.winntmag.com/Articles/Content/208_01.html?Key=Clustering], June 1997.

32. Microsoft Corporation, *Writing Microsoft Cluster Server (MSCS) Resource Dynamic-Link Libraries (DLLs) (White Paper)*, 1997.

33. Cristian, F., "Reaching Agreement on Processor-Group Membership," *Distributed Computing*, v. 4, pp.175-187, 1991.

34. Pradhan, D. K., "Recoverable Mobile Environment Design and Trade-off Analysis," *Proceedings of IEEE Annual Symposium on Fault Tolerant Computing,* pp. 16-25, June 1996.

35. Xie, G., Hensgen, D., Kidd, T., and Yarger, J., "SAAM: An Integrated Network Architecture for Integrated Services," *1998 Sixth International Workshop on Quality of Service,* pp. 117-126, May 1998.

36. Vrable, J. D. and Yarger, W. J., *The SAAM Architecture: Enabling Integrated Services*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1999.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Genelkurmay Baskanligi.....1
   Personel Baskanligi
   Bakanliklar
   Ankara, TURKEY

3. Kara Kuvvetleri Komutanligi.....1
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

4. Kara Kuvvetleri Komutanligi.....1
   Kutuphanesi
   Bakanliklar
   Ankara, TURKEY

5. Kara Harp Okulu.....2
   Kutuphanesi
   Dikmen
   Ankara, TURKEY

6. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

7. Chairman, Code CS.....1
   Naval Postgraduate School
   Monterey, CA 93943-5101

8. Prof.. Geoffrey Xie, Code CS/Xg.....1
   Naval Postgraduate School
   Monterey, CA 93943-5100

9. Prof. James Brett Michael, Code CS/Mj.....1
Naval Postgraduate School
Monterey, CA 93943-5100

10. 1LT Efraim Kati.....1
Selanik cad. 9 / 22
Kizilay, 06420
Ankara, TURKEY

60 290NPG 2390
TH
6/02 22527-200 NLB

DUDLEY KNOX LIBRARY
3 2768 00403124 5